



**UNIVERSIDAD  
DE GRANADA**

TRABAJO FIN DE MÁSTER

**Desarrollo de un videojuego  
para navegadores sobre  
WebGL**

---

**Autor**

Cristian Félix Chaves Muñoz

**Director/es**

Carlos Ureña almagro



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE  
TELECOMUNICACIÓN

—  
Granada, 12 de septiembre de 2024





# Desarrollo de un videojuego para navegadores sobre WebGL

---

## Autor

Cristian Félix Chaves Muñoz

## Director/es

Carlos Ureña almagro



# **Desarrollo de un videojuego para navegadores sobre WebGL:**

Cristian Félix Chaves Muñoz

**Palabras clave:** Aplicaciones web, WebGL, Videojuego

## **Resumen**

La evolución de la tecnología web ha permitido en estos últimos años que su utilización se haya expandido, de servir simplemente para el desarrollo de páginas web a permitir el desarrollo aplicaciones de propósito general. Se trata de una buena opción frente a otras tecnologías gracias a su sencillez y a la gran integración con distintos sistemas debido al uso del navegador web.

A raíz de lo anterior han ido surgiendo nuevas formas de interactuar con la web, siendo una de las más significativas la incorporación de escenas renderizadas 3D. Esto ha abierto un amplio abanico de posibilidades al ofrecer nuevas experiencias web, concretamente, en el desarrollo de videojuegos orientados a web, gracias a la utilización de WebGL, que es una librería gráfica para el navegador que permite hacer uso de la unidad de procesamiento gráfico para la generación de escenas 3D de alto rendimiento.

Este trabajo fin de máster tiene como objetivo el desarrollo de un pequeño videojuego 3D que logre ser multiplataforma gracias al uso de tecnologías web, además de lograr una ejecución eficiente aprovechando la potencia ofrecida por los procesadores gráficos, para ello se usan las siguientes tecnologías:

- HTML5, lenguaje de marcas orientado a web que permite definir un espacio de renderizado “canvas” que mostrará las escenas renderizadas del juego.
- JavaScript, lenguaje de programación interpretado orientado a web.
- TypeScript, lenguaje de programación que amplía la sintaxis de JavaScript añadiendo la declaración en los tipos de datos y la detección de errores en tiempo de compilación, siendo de gran utilidad en desarrollos software complejos.
- WebGL, librería gráfica orientada a web, desarrollada a partir de OpenGL, que permite el renderizado de escenas 3D aprovechando la potencia de cálculo del procesador gráfico o GPU.

A lo largo de este documento se hace un estudio de la librería WebGL con el fin de lograr renderizar una escena 3D, tratando todos los elementos imprescindibles que lo conforman:

- Creación y enlazado de buffers con los datos de los modelos 3D para ser renderizados por la GPU.
- Creación de los programas que le indicarán a la GPU cómo han de ser renderizados los modelos (Vertex Shader y Fragment Shader).
- El uso de las matrices necesarias para poder colocar los modelos en la escena, agregar la perspectiva y definir una cámara.
- Sistemas de luces que iluminarán las escenas.
- Uso de texturas en los modelos 3d.

Con todo esto, se crea una pequeña librería que mediante una interfaz permite dar acceso a todas las funciones anteriormente mencionadas, consiguiendo un desarrollo diferenciador entre la parte de WebGL y la parte del juego. Con esto se consigue un bajo acoplamiento y una alta cohesión.

Tras este desarrollo, se procede a la descripción del juego, que se ha elegido implementar. Se trata de un juego RPG de exploración de mazmorras o dungeon crawler en inglés, en primera persona, usando una temática fantástica medieval oscura, donde el usuario tendrá que avanzar entre las distintas mazmorras generadas aleatoriamente, hasta alcanzar la última de ellas. Durante su exploración el jugador debe hacer frente a enemigos típicos de este género como ogros y murciélagos, para lo se pone a su disposición una espada y un arco, así como unos objetos consumibles, pociones y flechas, repartidos en el mapa.

A medida que el protagonista avanza en las mazmorras los enemigos se hacen más fuertes, por ello es imprescindible que el jugador aumente el nivel de su personaje para lograr salir victorioso. Durante desarrollo software del juego se hace uso de la librería anteriormente creada para producir el renderizado, como la carga y posicionamiento de los modelos 3D y la creación y ubicación de la cámara y luces posicionales existentes.

Posteriormente se aborda el diseño gráfico del juego, mostrando:

- Los elementos 3D que conforman la escena (enemigos, paredes, suelos, armas, etc..) y de sus animaciones.
- La interfaz del usuario (UI) y de los elementos que la conforman.

Posteriormente, con el fin de hacer el juego multiplataforma se hace un análisis de las peculiaridades de los dispositivos para los que va destinado el juego, estudiando los elementos de entrada que estos ofrecen para que el usuario interactúe con el juego, el tipo de interfaz y cómo se deben mostrar sus elementos según el tamaño de la pantalla, con el fin de evitar, entre otras cosas, que en las pantallas pequeñas la información que se muestre no llegue a ser legible.

Por último, se realiza una prueba de rendimiento entre distintos navegadores web de algunos dispositivos, tomando como referencia el número de fotogramas generado por segundo (FPS) y el valor que resulte se tomará para medir el rendimiento y comprobar si existen diferencias entre ellos.



# **Development of a browser-based video game using WebGL:**

Cristian Félix Chaves Muñoz

**Keywords:** Web applications, WebGL, Video game

## **Abstract**

The evolution of web technology has allowed in recent years that its use has expanded, from simply serving for the development of web pages to allow the development of general purpose applications.

It is a good option compared to other technologies thanks to its simplicity and the great integration with different systems due to the use of the web browser.

As a result of the above, new ways of interacting with the web have emerged, one of the most significant being the incorporation of 3D rendered scenes. This has opened a wide range of possibilities by offering new web experiences, specifically in the development of web-oriented video games, thanks to the use of WebGL, which is a graphics library for the browser that allows the use of the graphics processing unit for the generation of high-performance 3D scenes.

This master's final project aims to develop a small 3D video game that can be multiplatform thanks to the use of web technologies, in addition to achieving an efficient execution taking advantage of the power offered by the graphics processors, for this the following technologies are used:

- HTML5, a web-oriented markup language that allows you to define a “canvas” rendering space that will display the rendered scenes of the game.
- JavaScript, a web-oriented interpreted programming language.
- TypeScript, a programming language that extends JavaScript syntax by adding data type declaration and compile-time error detection, which is very useful in complex software development.
- WebGL, a web-oriented graphics library, developed from OpenGL, which allows the rendering of 3D scenes using the computing power of the graphics processor or GPU.

Throughout this document a study of the WebGL library is made in order to render a 3D scene, dealing with all the essential elements that conform it:

- Creation and linking of buffers with 3D model data to be rendered by the GPU.
- Creation of the programs that will tell the GPU how the models are to be rendered (Vertex Shader and Fragment Shader).
- The use of the necessary matrices to be able to place the models in the scene, add perspective and define a camera.
- Lighting systems that will illuminate the scenes.

With all this, a small library is created that by means of an interface allows to give access to all the functions previously mentioned, achieving a differentiating development between the WebGL part and the game part. With this a low coupling and a high cohesion is achieved.

After this development, we proceed to the description of the game, which has been chosen to implement. It is an RPG game of dungeon crawler exploration, in first person, using a dark medieval fantasy theme, where the user will have to advance through the various randomly generated dungeons, until reaching the last of them. During his exploration the player must face typical enemies of this genre such as ogres and bats, for which he has at his disposal a sword and a bow, as well as consumable items, potions and arrows, scattered on the map.

As the protagonist progresses through the dungeons, the enemies become stronger, so it is essential for the player to increase the level of his character in order to be victorious. During the software development of the game, the previously created library is used to produce the rendering, such as the loading and positioning of 3D models and the creation and placement of the existing camera and positional lights.

Subsequently, the graphic design of the game is addressed, showing:

- The 3D elements that make up the scene (enemies, walls, floors, weapons, etc.) and their animations.
- The user interface (UI) and the elements that make it up.

Subsequently, in order to make the game multiplatform, an analysis is made of the peculiarities of the devices for which the game is intended, studying the input elements that they offer for the user to interact with the game, the type of interface and how its elements should be displayed according to the size of the screen, in order to avoid, among other things, that on small screens the information displayed is not legible.

Finally, a performance test is run between different web browsers on some devices, taking as reference the number of frames generated per second (FPS) and the resulting value will be taken to measure the performance and check if there are differences between them.

---

Yo, **Cristian Félix Chaves Muñoz**, alumno del **MÁSTER DE DESARROLLO DEL SOFTWARE**, con DNI **45102966N**, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Máster en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: *Cristian Félix Chaves Muñoz*

Granada a 12 de septiembre de 2024.



---

D. **Carlos Ureña almagro**, Profesor del Área de Lenguajes y Sistemas Informáticos del Departamento Lenguajes y Sistemas Informáticos de la Universidad de Granada.

**Informa:**

Que el presente trabajo, titulado *Desarrollo de un videojuego para navegadores sobre WebGL*, , ha sido realizado bajo su supervisión por **Cristian Félix Chaves Muñoz**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 12 de septiembre de 2024.

**El director:**

**Carlos Ureña almagro**



# **Agradecimientos**

Agradecer a mi familia su apoyo incondicional.



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	1
1.2. Objetivos . . . . .	2
1.3. Estructura del trabajo . . . . .	3
<b>2. Estado del arte</b>	<b>5</b>
2.1. Introducción a WebGL . . . . .	7
2.1.1. Shaders y GLSL . . . . .	14
2.1.2. Luces . . . . .	17
2.1.3. Cámara . . . . .	21
2.1.4. Texturas . . . . .	28
<b>3. Desarrollo del TFM</b>	<b>33</b>
3.1. Tecnologías del proyecto . . . . .	33
3.2. Capa software para aplicaciones WebGL . . . . .	36
3.3. Análisis y Diseño del software de Videojuego . . . . .	39
3.3.1. Requisitos funcionales . . . . .	39
3.3.2. Lógica del juego . . . . .	40
3.3.3. Generación de las mazmorras . . . . .	41
3.3.4. Desarrollo software del juego . . . . .	43
3.4. Diseño visual y modelado . . . . .	46
3.5. Portabilidad . . . . .	53
<b>4. Resultados y discusión</b>	<b>63</b>
<b>5. Conclusiones</b>	<b>65</b>
<b>Bibliografía</b>	<b>68</b>



# Índice de figuras

2.1.	WebGL Pipeline . . . . .	9
2.2.	Interpolación color triángulo . . . . .	10
2.3.	Esquema de enlace atributos . . . . .	12
2.4.	Tipos de luces . . . . .	17
2.5.	Cálculo normal . . . . .	18
2.6.	Modelo Phong . . . . .	19
2.7.	reflexion . . . . .	19
2.8.	Dragrama lambert . . . . .	20
2.9.	Dragrama especular . . . . .	20
2.10.	Perspectiva . . . . .	22
2.11.	Matriz model transform . . . . .	23
2.12.	Matriz view transform . . . . .	23
2.13.	Frustum . . . . .	24
2.14.	Ejemplo Normal errónea . . . . .	25
2.15.	Fujo de la cámara . . . . .	26
2.16.	matriz de la cámara . . . . .	27
2.17.	Ejemplo textura . . . . .	29
2.18.	Ejemplo mipmap . . . . .	29
2.19.	Ejemplo de rasterización . . . . .	30
2.20.	Filtro linear . . . . .	31
2.21.	Filtro nearest . . . . .	31
2.22.	Filtro repeat . . . . .	32
2.23.	Ejemplo mirrorRepeat . . . . .	32
3.1.	Diagrama de WebGL . . . . .	38
3.2.	Ejemplo de partición espacio Binario . . . . .	42
3.3.	Partición espacio Binario . . . . .	42
3.4.	Diagrama de clases del juego . . . . .	46
3.5.	modelo ogro . . . . .	47
3.6.	modelo murciélagos . . . . .	47
3.7.	modelo ogro andando . . . . .	48
3.8.	modelo ogro atacando . . . . .	48
3.9.	modelo ogro dañado . . . . .	48

3.10. modelo murciélagos volando . . . . .	48
3.11. modelo murciélagos atacando . . . . .	48
3.12. modelo muro . . . . .	49
3.13. modelo suelo . . . . .	49
3.14. modelo techo . . . . .	49
3.15. Ejemplo de escena . . . . .	50
3.16. modelo poción . . . . .	50
3.17. Modelo de flecha . . . . .	50
3.18. modelo escaleras . . . . .	51
3.19. Modelo de espada . . . . .	51
3.20. modelo del arco . . . . .	51
3.21. Interfaz PC . . . . .	52
3.22. Diagrama de pantallas del juego . . . . .	53
3.23. Imagen renderiza distintos fov . . . . .	55
3.24. Interfaz PC . . . . .	57
3.25. Interfaz móvil . . . . .	57
3.26. Interfaz móvil log . . . . .	58
4.1. imagen del juego pc . . . . .	64
4.2. imagen del juego móvil . . . . .	64

# Índice de tablas

2.1. tipos Datos GLSL . . . . .	16
3.1. Evolución estándar ECMAScript . . . . .	34
3.2. tabla resultados PC . . . . .	59
3.3. portatil . . . . .	59
3.4. tabla resultados samsung . . . . .	59
3.5. tabla resultados lenovo . . . . .	60
3.6. tabla resultados Odin . . . . .	60
3.7. tabla resultados iphone . . . . .	61



# Capítulo 1

## Introducción

El siguiente trabajo fin de máster tiene como propósito aplicar tecnologías web para el desarrollo de aplicaciones interactivas gráficas 2D o 3D multiplataformas.

Hoy en día es común encontrarse con una gran diversidad de dispositivos en el mercado, cada uno con sus propias características, por lo que lo ideal sería desarrollar una aplicación que pueda ejecutarse en el mayor número de dispositivos posibles con el consiguiente ahorro de tiempo y esfuerzo para el desarrollador y una de las mejores opciones que se puede presentar es realizar una aplicación web que se ejecute sobre el navegador, en vez de múltiples para cada dispositivo, lo que constituye el centro de este trabajo.

Las tecnologías web han ido evolucionando con el paso del tiempo adquiriendo unas características idóneas para su utilización tanto en el desarrollo de aplicaciones multiplataformas como para desarrollar aplicaciones interactivas que usen gráficos 2D o 3D gracias a su integración con librerías gráficas como webGL que permite aprovechar la potencia de cálculo de las unidades de procesamiento gráfico (GPU). Estos puntos son puesto en práctica a lo largo de este trabajo fin de máster, obteniendo como resultado un juego que demuestra que es factible el desarrollo de una única aplicación interactiva gráfica sin tener que entrar en detalles de desarrollo propios de cada dispositivo, como lenguajes o frameworks.

### 1.1. Motivación

Las principales motivaciones que han llevado a la realización de este trabajo de fin de máster han sido el aprendizaje y mejora de los conocimientos y conceptos sobre el uso de tecnologías web para desarrollar aplicaciones multiplataformas y el estudio de la librería gráfica WebGL para llevar el renderizado 3D a las aplicaciones web permitiendo el desarrollo en este caso

de un videojuegos orientado al navegador.

## 1.2. Objetivos

El principal objetivo del proyecto consiste en el desarrollo de una aplicación 3D interactiva que sea capaz de ejecutarse en múltiples dispositivos gracias a la utilización de tecnologías web y de librerías gráficas que aprovechan las capacidades de cómputo de las GPU. Para lograr este objetivo general se propone los siguientes objetivos específicos que permitirán la realización de este trabajo fin de máster, como se detalla a continuación:

1. Estudio de tecnologías web para el desarrollo de aplicaciones multiplataformas
  - Estudio de HTML5
  - Estudio de JavaScript
  - Estudio de TypeScript
2. Estudio de librería gráfica
  - Estudio de WebGL
3. Estudio sobre portabilidad
  - Características de entrada de los dispositivos
  - Adaptación de la interfaz a las características de los dispositivos.
  - Diferencias de rendimiento entre navegadores.
4. Desarrollo de un juego 3D multiplataforma.
  - Diseño conceptual del juego
  - Desarrollo de juego aplicando tecnologías web TypeScript y WebGL
  - Implementación y pruebas del juego.

En primer lugar, se realiza un estudio sobre las posibles tecnologías web, justificando las elecciones realizadas, y, como se podrá ver en el resto de los capítulos, se ha utilizado HTML5 junto con TypeScript, que es un superconjunto de JavaScript que ofrece tipado de datos. Posteriormente se realiza el estudio sobre la librería WebGL la cual es una interfaz de programación de aplicaciones (API) de una de las librerías gráficas más populares, OpenGL. A continuación, se hace un estudio de las características de los principales dispositivos que nos podemos encontrar en el mercado para los cuales se desarrolla la aplicación, en este estudio se ha tenido en cuenta,

por ejemplo, la forma que tiene de recibir las entradas: teclado, pantalla, mando de juego, etc.. ya que son de interés para lograr obtener un producto multiplataforma. Por último, se aplica todo lo anterior al desarrollo de un juego simple.

### 1.3. Estructura del trabajo

La estructura en la que queda organizada este trabajo fin de máster es el siguiente:

En el capítulo 2 (Estado del arte) se describen las tecnologías relacionadas con este TFM, como las aplicaciones web, librerías gráficas existentes y frameworks de desarrollo de juegos más conocidos, por último se hará un repaso de los conceptos principales de WebGL y de la informática gráfica.

En el capítulo [3 \(Desarrollo del TFM\)](#) se aborda la mayor parte del proyecto, donde se tratarán los siguientes puntos:

- Tecnologías de proyecto, donde se introduce brevemente a las tecnologías de las que se ha hecho uso para el desarrollo de la aplicación.
- Capa software para aplicaciones WebGL, punto donde se lleva a cabo la explicación de la capa software que contiene la interfaz con todos los elementos necesarios para renderizar la escena.
- Desarrollo del juego, donde se detalla todos los aspectos relevantes del juego desarrollado.
- Diseño, en este punto se detalla los elementos visuales desarrollados para el proyecto del juego, desde los modelos 3D hasta las interfaces del usuario.
- Portabilidad, último punto del desarrollo de este TFM, donde se realiza un estudio de los distintos dispositivos para los cuales va destinado el juego, analizando sus características de entrada, tamaño de pantalla y otras características, finalizando en un estudio de rendimiento comparativo.

En el capítulo [4 \(Resultados y discusión\)](#) Se describen los objetivos que se han alcanzado en este trabajo.

Por último, en el capítulo [5 \(Conclusiones\)](#) se finaliza con algunas conclusiones a las que se ha llegado tras el desarrollo del proyecto.



# Capítulo 2

## Estado del arte

Para estudiar el contexto de este trabajo es necesario atender al concepto y evolución de aplicaciones web y librerías gráficas lo que se desarrolla a lo largo de los siguientes apartados.

### Aplicaciones Web o WebAPP

Al hablar de tecnologías web se hace referencia a aquellas tecnologías que permiten el procesamiento y mostrado de la información en un navegador web. En la actualidad es común referenciar al lenguaje de marcado HTML junto con el lenguaje de programación JavaScript.

HTML es el lenguaje de marcado por excelencia, utilizado tanto en páginas web como en aplicaciones web. Con el paso del tiempo se le han ido añadiendo características tales como la posibilidad de incorporar un elemento “Canvas” que permite ser usado para el contexto de pintado en aplicaciones gráficas, elemento interesante del que se hace uso en este trabajo.

JavaScript es un lenguaje de programación ligero interpretado, capaz de ser ejecutado en el navegador, dándole características de portabilidad entre distintos sistemas, otro elemento importante en nuestro proyecto. Pero debido a que no tiene tipado de datos para las variables, puede darse el caso de errores de asignación en tiempo de ejecución, para evitar este problema se utiliza TypeScript, el cual es un supertipo de JavaScript, que añade tipado de datos, solventando errores de asignación al detectarse en tiempo de compilación, facilitando y ahorrando tiempo en la búsqueda del error.

### Librerías gráficas

Existen una gran variedad de librerías gráficas que permiten desarrollar aplicaciones 2D y 3D. Entre las más importantes se encuentran:

- Para aplicaciones nativas:

- OpenGL: Según su sitio web oficial [1], OpenGL es una interfaz de desarrollo de aplicaciones (API) de bajo nivel que permite desarrollar aplicaciones 2D y 3D independiente del sistema de ventanas y del sistema operativo, ampliamente utilizado en la industria del desarrollo de aplicaciones gráficas.
- Direct3D: API propietaria de Microsoft que según su página oficial [2], Direct3D es una API de bajo nivel destinada a gráficos 2D y 3D, que permite aprovechar las características de las GPU para realizar operaciones altamente paralelas abstrayendo las implementaciones de cada GPU.
- Vulkan: De código abierto, es considerado la evolución de OpenGL, según se detalla en el sitio web de gpuOpen [3], Vulkan es una API de gráficos 3D que en comparación con OpenGL, reduce la sobrecarga en la GPU y añade capacidades nuevas, tales como soporte para teléfonos inteligentes (smartPhones) y tabletas Android, siendo multiplataforma para sistemas Windows y Linux.
- Metal: API propietaria de Apple para la gama de sus productos (Ipad, Iphone, Mac,...), según en su sitio web [4] permite la generación de gráficos optimizado para las plataformas Apple gracias al uso de un lenguaje de shading propio ”Metal Shading Language”, maximizando la eficiencia en los programas gráficos desarrollados.

- Para aplicaciones web:

- WebGL: Librería gráfica gestionado por el grupo ”Khronos Group”, permite renderizar escenas 3D en navegadores web de manera eficiente gracias a la utilización de la GPU para el renderizado, tiene sus orígenes en OpenGL ES 2.0, esta es la librería usada en este proyecto.
- Three.js: Bajo licencia MIT, es una biblioteca de JavaScript que según su sitio web oficial [5], utiliza WebGL para dibujar gráficos 3D. Ofrece la ventaja de no tener que entrar en detalles a bajo nivel de implementación ofreciendo fácil manejo sobre cosas como escenas, luces, sombras, materiales, texturas, matemáticas 3D, lo cual debería ser tratado si se utilizará webGL directamente.

A continuación, se presenta algunas de las bibliotecas y frameworks más populares para el desarrollo de juegos:

- SDL: Según se detalla en su sitio web [6], Simple DirectMedia Layer es una biblioteca de desarrollo multiplataforma escrita en C, diseñada

para proporcionar acceso de bajo nivel a audio, teclado, ratón, joystick y hardware gráfico a través de OpenGL y Direct3D. SDL es compatible oficialmente con Windows, macOS, Linux, iOS y Android.

- LibGDX: En su sitio web [7] describen LibGDX como un marco de desarrollo de juegos Java multiplataforma basado en OpenGL funcionando en diversidad de sistemas como Windows, Linux, macOS, Android, su navegador e iOS. Tiene una licencia Apache 2.0 y es mantenida por la comunidad.
- LWJGL: Biblioteca java para el desarrollo de videojuegos. Según la información proporcionada por su sitio web oficial [8], LWJGL (Light-weight Java Game Library ) es una biblioteca Java multiplataforma que permite utilizar APIs populares para el desarrollo de gráficos (OpenGL, Vulkan), audio (OpenAL) y computación paralela (OpenCL), proporcionando acceso de bajo nivel.
- Phaser: Es un framework para el desarrollo de juego en HTML5 para navegadores web. Según la información proporcionada por su web oficial [9], utiliza tecnologías web para crear juegos compatibles con ordenadores de escritorio o móviles, o aplicaciones capaces de ejecutar juegos web, como Discord, Facebook, etc.. Phaser se centra principalmente en el desarrollo web para juegos 2D, sin incluir la posibilidad de 3D. Phaser está desarrollado en JavaScript.
- Pygame: Es un conjunto de módulos Python diseñados para escribir videojuegos. Según en su sitio web oficial [10], Pygame añade una capa de abstracción a la biblioteca SDL, permitiendo crear juegos y programas multimedia sencillos. Pygame es muy portable llegando a ser ejecutado en casi todas las plataformas y sistemas operativos. Cuenta con licencia LGPL.

## 2.1. Introducción a WebGL

En esta sección se tratan los principales conceptos de la librería WebGL, desde la tubería de renderizado, los buffers de datos, hasta como referenciarlos para que sean usados por el programa encargado del renderizado, para ello, se atiende a lo descripción dada por los autores Ghayour y Cantor en su libro [11].

WebGL es una librería gráfica que permite renderizar escenas 3D en navegadores web de manera eficiente gracias a que utiliza la GPU (graphics processing unit) para el renderizado, tiene sus orígenes en OpenGL ES 2.0 el cual es una especificación de OpenGL para dispositivos móviles, con la idea de alcanzar a cualquier tipo de dispositivo logrando el renderizado en tiempo

real en navegadores web. Es compatible con la gran mayoría de navegadores del mercado abriendo su uso a una inmensa cantidad de dispositivos, tanto de escritorio como móviles.

En WebGL el renderizado se realiza en la máquina cliente , los elementos que forman parte de la escena se descargan desde un servidor y el procesamiento necesario para generar la imagen se realiza en el hardware del cliente.

WebGL presenta las siguientes características:

- Uso de JavaScript como lenguaje de programación, lenguaje propio en entornos web, además de la posibilidad de utilizar otras librerías como Jquery, React y Angular.
- Gestión automática de la memoria, donde la asignación y desasignación se hace automáticamente gracias al uso de JavaScript.
- Generalización, los navegadores web con capacidades de JavaScript están instalados en teléfonos inteligentes y tabletas, por lo que se puede aprovechar WebGL en un inmenso número de dispositivos, tanto móviles como de escritorio.
- Rendimiento eficiente gracias al uso del hardware de la GPU.

### **Elementos de una aplicación WebGL**

Los elementos comunes de una aplicación con WebGL son:

- Canvas: Elemento HTML5 que permite definir el espacio dentro de la web donde se renderiza la escena, se puede acceder a él por medio del DOM.
- Objetos: Entidades 3D que componen la escena. Estas entidades se representan como una malla de triángulos almacenándose en buffers para ser dispuestos para la tarjeta gráfica.
- Luces: WebGL permite usar Shaders para modelar las luces de una escena.
- Cámara: Elemento indispensable para poder explorar la escena 3d.

### **WebGL versión 2**

Para el desarrollo de este trabajo se hace uso WebGL en su versión 2, el cual es una actualización de WebGL, basada en OpenGL ES 3.0. entre las mejoras que ofrece con su versión anterior destacan:

- El uso de VAOs (Vertex Array Objects) está disponible sin necesidad de añadir complementos.
- Posibilidad de saber el tamaño de la textura desde el Shader.
- Mayor número de formatos de texturas.
- Soporte para texturas tridimensionales.
- Permitir texturas que no son potencia de dos para generar mipmap.
- Adición de la funciones inverse y transpose en WebGL2 GLSL 300 para calcular la matriz inversa y la traspuesta respectivamente en el shader.

### Tubería de renderizado en WebGL

WebGL es un motor de rasterización, dibuja puntos, líneas y triángulos según un par de código que le proporciona a la GPU, siendo estas el Vertex Shader y el Fragment Shader, las cuales están escritas en un lenguaje de programación similar a C llamado GLSL (GL Shader Language).

En la figura 3.1 se muestra una versión simplificada del proceso de renderizado de WebGL:

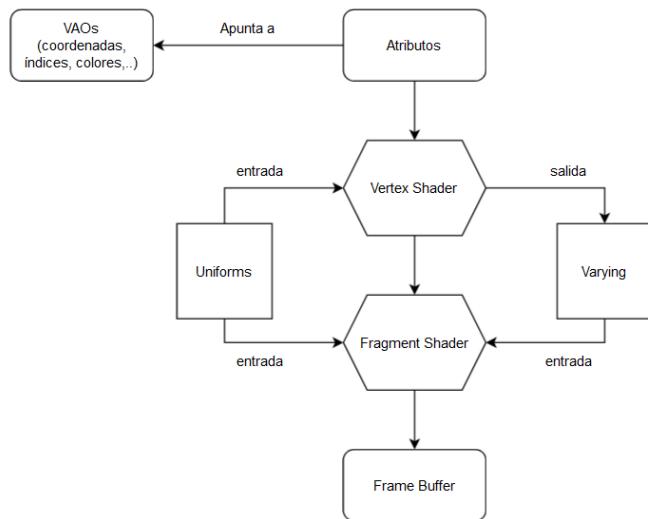


Figura 2.1: Esquema de la tubería de renderizado de WebGL

- Vertex Buffer Objects (VBOs): contienen los datos de la geometría a dibujar, como por ejemplo las coordenadas de los vértices, normales de vértices, colores, coordenadas de texturas, etc..
- Vertex Shader: Encargado de manipular los datos de cada vértice, como coordenadas, normales, colores, etc.
- Fragment Shader: El objetivo principal de Fragment Shader es calcular el color de los píxeles individuales o fragmentos de los modelos que se verán en pantalla.

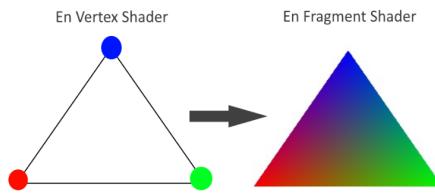


Figura 2.2: Ejemplo de interpolación para cacular el color del triángulo

- FrameBuffer: Una vez procesados los fragmentos por el Fragment Shader se almacenan en un Buffer bidimensional el cual conforma la imagen 2D que se muestra en pantalla.
- Attributes: Variables de entrada que se utilizan en el Vertex Shader. Dado que Vertex Shader se invoca por cada vértice, los atributos serán diferentes cada vez.
- Varying: Se utilizan para pasar los datos del Vertex Shader al Fragment Shader.

### Renderizado en WebGL:

Para la representación de la geometría de cualquier objeto 3D es necesario indicar al menos:

- Vértices: Los puntos donde confluyen tres o más aristas de los objetos 3D, donde cada vértice está representado por tres puntos en coma flotante correspondientes a las coordenadas x, y, z del vértice. WebGL requiere de todos los vértices del modelo durante el proceso de renderizado, por lo cual todos los vértices deben ser proporcionados en una matriz que con posterioridad se usará para construir el buffer de vértices.

- Índices: Permite decirle a WebGL como conectar los vértices para producir la superficie, al igual que con los vértices, también se almacenan en una matriz que serán usados en el renderizado mediante un buffer.
- Normales: Indican la orientación de cada vértice, necesario para la interacción con las luces y la visualización de las texturas.
- Colores: Especifican el color del vértice, se utiliza como representación del color el sistema RGB (Red, Green, Blue) y como adicional la componente alfa que indica la transparencia. El color de la primitiva triángulo se hace por interpolación de sus tres vértices.

Para que los anteriores elementos puedan ser pasados a la unidad de procesamiento gráfico para su interpretación es necesario pasarlos a unos buffers especiales denominados Vertex Buffer Objects (VBO). Para hacer uso de los VBO de WebGL hay que seguir los siguientes pasos:

1. Crear un nuevo buffer mediante el método `createBuffer()`
2. Vincularlo para indicar que es el buffer actual a usar, esto se realiza con la función `bindBuffer( tipoBuffer, buffer)`, donde los valores de tipoBuffer son:
  - *ARRAY\_BUFFER*, si son vértices
  - *ELEMENT\_ARRAY\_BUFFER*, si son índices.

Y el parámetro buffer es el buffer creado anteriormente.

3. Pasar los datos al buffer mediante la función `bufferData (tipoBuffer, datos, tipo)`, donde tipoBuffer es como en el anterior método, datos es el array con los datos y tipo puede ser uno de estos tres valores.
  - *STATIC\_DRAW*, Si los valores del buffer no serán cambiados.
  - *DYNAMIC\_DRAW*, Si los valores del buffer serán cambiados.
  - *STREAM\_DRAW*, Si los valores del buffer cambian cada ciclo de renderizado.

WebGL requiere de unos arrays proporcionados por JavaScript para que los datos del buffer puedan procesarse de forma binaria para lograr aumentar el rendimiento del procesamiento de la geometría, estos arrays son: *Int8Array*, *Uint8Array*, *Int16Array*, *Uint16Array*, *Int32Array*, *Uint32Array*, *Float32Array* y *Float64Array*. Es importante tener en cuenta que las coordenadas de los vértices pueden ser flotantes, pero los índices son siempre números enteros.

Una vez creado los VBOs, el siguiente paso es asociar los buffer a los atributos del Vertex Shader.

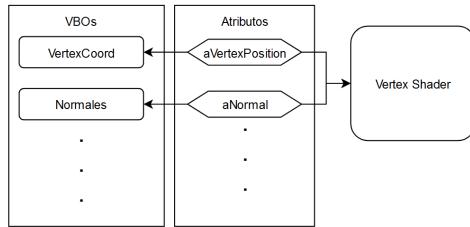


Figura 2.3: Esquema de enlazado entre los VBO y los atributos para ser consumidos por el Vertex Shader

Para ello:

1. Apuntar un atributo al VBO actualmente vinculado, la función WebGL que permite vincular los VBOs a los atributos es:

```
1 vertexAttribPointer(referencia_atributo, tamaño, tipo, normalizar, paso, desplazamiento)
```

Donde:

- *referencia\_atributo*: especifica el índice del atributo que vamos a asignar al buffer actualmente vinculado.
- *tamaño*: número de valores por vértice.
- *tipo*: Tipo de datos almacenados en el buffer actual, puede ser: *FIXED*, *BYTE*, *UNSIGNED-BYTE*, *FLOAT*, *SHORT* O *UNSIGNED-SHORT*.
- *normalizar*: valor booleano que indica si los valores se normalizaron en un determinado rango, por ejemplo: los datos tipos *BYTE* y *SHORT*, se normalizará entre el rango [-1, 1] si el valor es true.
- *paso*: Los bytes entre el comienzo de Atributos de vértice consecutivos, si es 0, entonces estamos indicando que los elementos se almacenan secuencialmente en el buffer.
- *desplazamiento*: La posición en el búfer. a partir del cual comenzaremos a leer valores para el atributo correspondiente. Generalmente se establece en 0 para indicar que comenzaremos a leer valores del primer elemento del búfer.

2. Habilitar el atributo: Posteriormente será necesario activar el atributo del Vertex Shader, para ello usamos la función:

```
1 enableVertexAttribArray(referencia_atributo)
```

## Vertex Array Objects (VAO)

Otro elemento importante a la hora del renderizado es el Vertex Array Objects, el cual es un objeto que contienen uno o varios VBOs de un modelo a renderizar, su utilización está muy recomendada dado que permite aumentar el rendimiento, pues se evita hacer las múltiples llamadas a las funciones como `gl.bindBuffer()` en tiempo de renderizado por cada buffer asociado al modelo. Al crearse un VAO se le enlazan todos los buffers necesarios, lo cual se puede hacer durante la inicialización de la aplicación, haciendo que en tiempo de renderizado solamente haya que indicar el VAO a utilizar. Para la implementación de los VAO en WebGL versión 1 era necesario el uso de extensiones, pero a partir de la versión 2 ya son incorporados de forma nativa.

Una vez que hayamos definido el VAO que contiene los VBO y asignado los atributos ya se podrá renderizar la escena haciendo uso de las funciones `drawArrays` o `drawElements`.

## Funciones de dibujado

El último paso consiste en indicar a WebGL que renderice el VAO actual activo, para ello se hace uso de las funciones `drawArrays` y `drawElements`, estas permiten escribir en el framebuffer.

- `drawArrays` utiliza datos de vértices en el orden en que están definidos en el búfer para crear la geometría. El problema de usar este método es que para cada primitiva triángulo se tendrá que volver a indicar los vértices comunes, repitiendo coordenadas en el VBO, esto no es lo más óptimo, dado que cuantos más vértices duplicados, más llamadas tendrá en el Vertex Shader, afectando al rendimiento general. la definición de `drawArrays` es la siguiente:

```
1 gl.drawArrays(modo, primero, recuento)
```

Donde:

- *modo*: Representa el tipo de primitiva que vamos a renderizar. Los valores posibles para el modo son `gl.POINTS`, `gl.LINE_STRIP`, `gl.LINE_LOOP`, `gl.LINES`, `gl.TRIANGLE_STRIP`, `gl.TRIANGLE_FAN` y `gl.TRIANGLES`.
  - *primero*: Especifica el elemento inicial en las matrices habilitadas.
  - *recuento*: El número de elementos que se van a representar.
- `drawElements` permite usar el IBO para indicar a WebGL cómo se ha de representar la geometría, lo que permite que los vértices sólo se

procesan una vez, reduciendo la memoria usada como el procesamiento requerido, la definición de drawElements es la siguiente:

```
1 gl.drawElements(modo, conteo, tipo, desplazamiento)
```

Donde:

- *modo*: Representa el tipo de primitiva que vamos a renderizar. Los valores posibles para el modo son *gl.POINTS*, *gl.LINE\_STRIP*, *gl.LINE\_LOOP*, *gl.LINES*, *gl.TRIANGLE\_STRIP*, *gl.TRIANGLE\_FAN* y *gl.TRIANGLES*.
- *conteo*: especifica el número de elementos que se representarán.
- *tipo*: especifica el tipo de valores en los índices. Debe ser *UNSIGNED\_BYTE* o *UNSIGNED\_SHORT*, ya que manejamos índices (números enteros).
- *desplazamiento*: indica qué elemento del búfer será el punto de partida para el renderizado. Suele ser el primer elemento (valor cero).

### 2.1.1. Shaders y GLSL

Esta sección describe qué son los programas Shaders y cómo implementarlos, para ello se siguen las explicación de los autores Ghayour y Cantor en su libro [11] y de Shreiner et al. [12].

Los Shaders o sombreadores son un elemento clave a la hora de renderizar los modelos de la escena, y son unos programas que son ejecutados en la unidad de procesamiento gráfico (GPU). Existen varios tipos de Shader, entre los que destacan el Vertex shader y el Fragment Shader, los cuales son llamados secuencialmente en la generación de la imagen 2D.

- Vertex Shader, encargado de operar los vértices (traslaciones, rotaciones, etc..) de los VBO provistos en los atributos. El Vertex Shader es llamado por cada vértice. Una vez procesados los vértices se pasa al proceso de rasterización por la que se genera la imagen 2D.
- Shader Fragment, se encarga de dar color y asignar la textura a cada punto de la imagen o fragmento generados por el rasterizador.

Dado que la llamada a estos Shader se hace por cada vértice y por cada píxel respectivamente se aprovecha las capacidad de computación en paralelo de las GPU.

Para crear estos Shader se usa el lenguaje OpenGL Shading Language (GLSL) que es un lenguaje de alto nivel similar a C++ que nos permite

definir la funcionalidad de cada Shader. WebGL incluye un compilador que permite coger el código fuente y generar el código que será provisto a la GPU.

Ejemplo:

```
1 #version 3 es
2 in vec4 vPosition;
3 void main()
4 {
5     gl_Position = vPosition;
6 }
```

Para que WebGL pueda compilar el código del programa escrito en GLSL este ha de ser pasado como una cadena de caracteres.

```
1 //Create a vertex shader object
2 this.vertShader = gl.createShader(gl.VERTEX_SHADER);
3
4 //Attach vertex shader source code
5 gl.shaderSource(this.vertShader, this.vertCode);
6
7 //Compile the vertex shader
8 gl.compileShader(this.vertShader);
```

Para poder usarlo es necesario de crear un Shader Program al cual se le asocian los Shaders Creados.

```
1 this.shaderProgram = gl.createProgram();
2
3 // Attach a vertex shader
4 gl.attachShader(this.shaderProgram, Vertex.vertShader);
5
6 // Link both programs
7 gl.linkProgram(this.shaderProgram);
```

Estando de esta forma listo para su uso cuando se invoque el método useProgram

```
1 gl.useProgram(this.shaderProgram);
```

## Estructura de los Shader

Al escribir un shader usando GLSL, la primera línea del Shader siempre tiene que ser la versión a utilizar, Actualmente la última versión es la 4, pero en WebGL que utiliza la misma que OpenGL ES es la versión 3. Siempre se le debe indicar el método main pues es el punto de partida de la ejecución del Shader.

Los tipos de datos tipos son los que se muestran en la tabla 2.1

Tipos base	Vectores 2D	Vectores 3D	Vectores 4D	Matrices
uint	uvec2	uvec3	uvec4	-
int	ivec2	ivec3	ivec4	-
float	vec2	vec3	vec4	mat2 mat2x2 mat 3x2 mat4x2 mat3 mat2x3 mat3x3 mat4x3 mat4 mat2x4 mat3x4 mat4x4
double	dvec2	dvec3	dvec4	dmat2 dmat2x2 dmat3x2 dmat4x2 dmat3 dmat2x3 dmat3x3 dmat4x3 dmat4 dmat2x4 dmat3x4 dmat4x4
bool	bvec2	bvec3	bvec4	-

Tabla 2.1: tipos Datos que soporta GLSL

Existen tres formas de pasar los datos a los Shader:

- Attributes o in: Datos procedentes de los Buffers o pasados por el shader previo, ejemplo: in vec3 vértice.
- Out: Datos que se pararán al siguiente shader.
- Uniforms: Valores pasados al Vertex Shader y que son los mismos para todos los vértices durante el dibujado, ejemplo de declaración: uniform vec4 vectorLuz

Los Shaders son ejecutados por la GPU cuando se invocan los métodos gl.drawArrays y gl.drawElements.

## Vertex Shader

El vertex Shader se encarga por lo general del cálculo para generar las coordenadas en el espacio de clip, el cual es el espacio de coordenadas a renderizar en pantalla mediante la transformación de los vértices del modelo, es invocado por cada vértice del modelo a renderizar.

El Vertex Shader tiene una variable global de salida donde se guarda el vértice en coordenadas homogéneas (x,y,z,w) que es necesario para realizar las operaciones de representar las coordenadas 3D a coordenadas 2D, lo que se conoce en inglés como normalized-device coordinates. Este proceso se explica en más detalle en el apartado de la cámara.

Dado al orden en el que se ejecutan los Shader, el vertex shader es el encargado de pasar los datos correspondientes al color y texturas al fragment shader.

## Fragment Shader

El fragment shader tiene la misma estructura que el Vertex Shader, siendo invocado por cada pixel, es el encargado de asignar el color al píxel que se está renderizando. Reciben los datos pasados por el Vertex Shader, generalmente datos del color, luces, índice de texturas, etc.. para realizar el cálculo del color. Además dispone de una variable uniforme de tipo sampler2D en el cual se puede almacenar los píxeles de la imagen que se esté usando como textura.

Se le debe definir una variable de salida con el color final del píxel que será utilizado en los pasos posteriores por WebGL.

### 2.1.2. Luces

Esta sección se habla de cómo se implementan las luces en un programa 3D, donde se describe los distintos tipos de luces, los modelos de sombreado y de luz de Phong, para ello se usa la explicación dada autores Ghayour y Cantor [11].

En la vida real podemos ver objetos debido a la reflexión de la luz en ellos, la iluminación de un objeto depende de su posición relativa a la fuente de luz, la orientación de su superficie y del material que está compuesto, todo esto es necesario ser modelado dentro de las aplicaciones para poder simular la luz dentro de las escenas.

#### Tipos de luces

Las fuentes de luz pueden ser de dos tipos, posicionales, cuando la luz se emite en todas las direcciones, como por ejemplo una lámpara, y direccionales, la luz se emite de en forma de rayos de luz paralelos en la misma dirección, como por ejemplo el sol. Las fuentes de luces posicionales son modeladas como puntos en el espacio y las direccionales como vectores que indican la dirección.

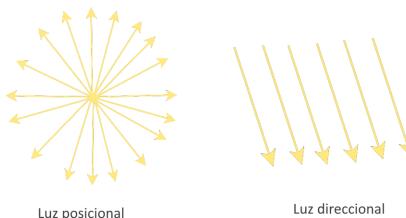


Figura 2.4: Imagen donde se aprecian los distintos tipos de luces

Para que estas luces puedan interaccionar con los modelos, necesitamos la información de la orientación de las caras de los modelos, esta es proporcionada por las normales, además del tipo del material.

### Normales

Las normales son vectores perpendiculares a la superficie del objeto a iluminar; representan la orientación de la superficie y son cruciales para modelar la interacción entre la fuente de luz y el objeto.

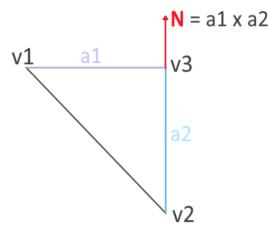


Figura 2.5: Cálculo de la normal de un vértice a través de sus aristas

### Materiales

Indican las propiedades del objeto, como color, brillo, transparencia, etc.. Esto permite definir cómo se verá el objeto, como por ejemplo con una superficie mate, metálica, cristalina, etc..

### Modelos de sombreado (Shading model) y modelos de luz (lighting model)

El modelo de sombreado hace referencia al tipo de interpolación que se llevará a cabo para obtener el color de cada fragmento (pixel) de la escena.

El modelo de luz determina como las normales, los materiales y las luces se combinan para producir el color final, para ello se usa los principios de la reflexión de la luz, por ello también se le llama modelo de reflexión.

Existen distintos tipos de modelos, en este trabajo se usan los modelos de sombreado y de luz de Phong.

### Modelo de sombreado de Phong

En el modelo de Phong el cálculo del color final se realiza en el Shader Fragment, para ello se ha de indicar la normal de cada vértice y mediante interpolación se calcula la normal de cada fragmento.

### Modelo de reflexión de Phong

El modelo de Phong presentado en Bui Tuong Phong en su doctorado en 1973, en ella especifica que el color resultante de un objeto ante una fuente de luz es la combinación de tres elementos: la luz ambiental, a la reflexión difusa del objeto y a la reflexión especular, tal y como se muestra en la figura 2.6

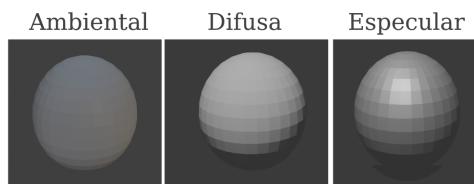


Figura 2.6: imagen de ejemplo en el uso de luces según el modelo de Phong

- Luz ambiental: Luz presente en la escena siendo el mismo para todos los fragmentos.
- Reflexión difusa: Produce una intensidad que es la misma en todas las direcciones de salida del punto de reflexión.
- Reflexión especular: Rayos de luz que rebotan sobre todo en ángulo opuesto a la luz que llega a la superficie, como ocurre en la superficie de un espejo.

Todas ellas se dan en términos de de color RGB.

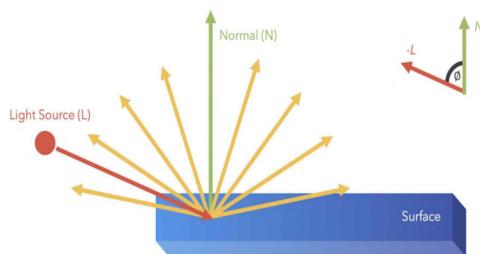


Figura 2.7: reflexion

### Cálculo de la luz ambiental

El cálculo de la luz ambiental solamente requiere que se multiplique la luz ambiental por la propiedad ambiental del modelo. Sea  $I_a$  el color ambiental:

$$Ia = C_l * C_m$$

Donde Cl y Cm son el color de la luz, y el color difuso del material.

### Cálculo de la luz difusa

Para el cálculo de la luz difusa se suele utilizar el modelo de reflexión de Lambert.

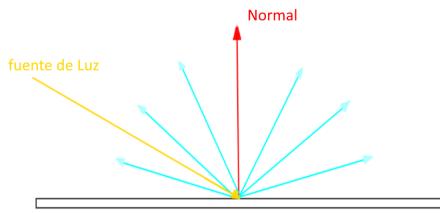


Figura 2.8: Diagrama de reflexión de la luz

Se calcula como el producto punto entre la normal de la superficie y el vector director de la luz en negativo, posteriormente este número se multiplica por el material del objeto y el color de la fuente de luz. Sea Id la luz difusa final:

$$Id = C_l C_m (-L \cdot N)$$

### Cálculo de la luz especular

El cálculo de la luz especular modelada como el producto punto entre dos vectores, el vector ojo y el vector de la luz reflejada.

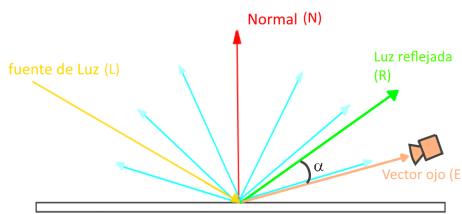


Figura 2.9: Diagrama del modelo de luz con los elementos necesarios para calcular la especularidad: vector fuente de luz, la normal del material, vector de la reflejada, y el vector hacia donde está orientada la cámara

Donde el vector ojo tiene origen en el fragmento y termina en la posición de la cámara y el vector luz reflejado es obtenido mediante la reflexión del vector luz sobre la normal de la superficie. Cuanto el producto punto es igual a 1 la cámara está enfocada en el punto de reflexión máxima, después el producto punto se le calcula el exponente por un número que representa que tan brillante es la superficie, y por último se multiplica por el color de la luz y por la componente especular del material. Sea  $I_s$  la luz especular final:

$$I_s = C_l C_m (R \cdot E)^n$$

Una vez hallados las propiedades ambientales, difusas y especular, se suman para obtener el color final de cada fragmento.

A continuación se presenta un ejemplo de cómo quedaría los cálculos anteriores en el Fragment Shader.

```

1 void main(void) {
2     vec3 L = normalize(direcci nLuz);
3     vec3 N = normalize(normal);
4     float lambert = dot(N, -L);
5     vec4 Ia = luzAmbiental * materialAmbiental;
6     vec4 Id = vec4(0.0 , 0.0 , 0.0 , 1.0);
7     vec4 Is = vec4(0.0 , 0.0 , 0.0 , 1.0);
8
9     if(lambert > 0.0){
10         //calculo luz difusa
11         Id = luzDifusa * materialDifuso * lambert;
12         vec3 E = normalize(vectorOjo);
13         vec3 R = reflect(L , N);
14
15         //calculo atributo especular
16         float especular = pow( max( dot (R , E), 0.0), brillo);
17         Is = luzEspecular * materialEspecular * especular;
18     }
19
20     vec3 color = Ia + Id + Is;
21
22     fragColor = vec4(color, 1.0);
23 }

```

### 2.1.3. Cámara

En esta sección se describen los conceptos principales para la implementación de una cámara en WebGL, basándose en las explicaciones de los autores Ghayour y Cantor en su libro [11].

WebGL no proporciona un objeto cámara que se pueda manipular, únicamente se dispone del espacio ofrecido por el canvas para renderizar, esté siempre orientado en el eje negativo de la coordenada z, por lo tanto, para simular el movimiento de una cámara se debe actualizar todos los objetos de

la escena con respecto a la nueva posición de la cámara. Para poder aplicar estas transformaciones en los elementos de la escena se utilizan un conjunto de matrices dimensión 4x4 las cuales codifican la transformación a desear, para poder trabajar con estas matrices se debe usar un sistema de coordenadas homogéneas en los vértices, pasando de tres componentes del espacio euclídeo ( $x,y,z$ ) a cuatro ( $x, y,z,w$ ) siendo la cuarta componente la componente de perspectiva. Las coordenadas homogéneas permiten trabajar en un espacio proyectivo.

Las coordenadas homogéneas hacen posible la resolución de sistemas de ecuaciones lineales donde cada ecuación representa una línea paralela con respecto a las otras del sistema, dado que las líneas intersectan en el infinito, mientras que en el espacio euclídeo esto no ocurre.

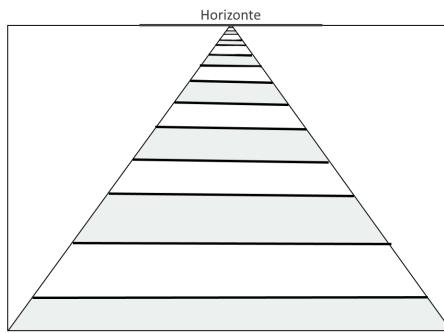


Figura 2.10: Ilustración de un punto de fuga en el horizonte

Es fácil pasar entre los dos sistemas:

- Para pasar de espacio homogéneo a no homogéneo, dividimos cada componente por la componente perspectiva.  $h(x, y, z, w) = v(x/w, y/w, z/w)$
- De espacio no homogéneo a homogéneo basta con añadir una nueva componente y darle valor 1.  $v(x, y, z) = h(x, y, z, 1)$ , si le hubiésemos dado un valor igual a 0 representaría un punto en el infinito.

Para poder implementar la cámara necesitamos realizar una serie de transformaciones los vértices de los modelos, estas transformaciones permiten pasar los modelos a distintas coordenadas, estas son:

### Transformación de modelado

Partimos de que los vértices de los modelos están especificados en el sistema de coordenadas del objeto y si queremos moverlo por la escena debemos

multiplicar estos por la matriz que codifica estas transformaciones , esta matriz se denomina en inglés model matrix. Cuando multiplicamos esta matriz por los vértices del modelo, obtenemos unas nuevas coordenadas para los vértices, estas nuevas coordenadas determinan la posición del modelo en el mundo de nuestra escena 3D, es decir a coordenadas del mundo.

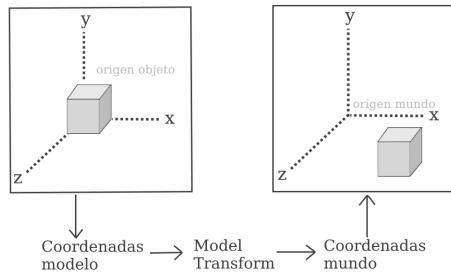


Figura 2.11: Ejemplo visual de cambio de coordenadas al aplicar la matriz de transformación del modelo

### Transformación de vista

La siguiente transformación es la transformación de vista, el cual permite cambiar el origen y la orientación del sistema de coordenadas a uno nuevo, siendo este nuevo origen donde la cámara estará localizada, permitiendo pasar de coordenadas del mundo a coordenadas de vista, estas transformaciones están codificadas en la matriz de vista o View matrix en inglés. Para lograr el cambio de coordenadas se debe multiplicar la matriz de vista por los vértices obtenidos durante la transformación del modelo a coordenadas del mundo.

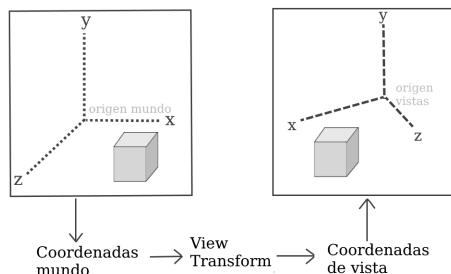


Figura 2.12: Ejemplo visual de cambio de coordenadas al aplicar la matriz de vista de la cámara

A la combinación mediante la multiplicación de las matrices anteriores se le conoce como la matriz de modelo-vista o model-view matrix en inglés.

### Transformación de proyección

La siguiente operaciones es la transformación de proyección, en la que se determina el espacio de vista que será renderizado y cómo será representado en la pantalla, a esta región se le conoce como frustum y está definida por seis planos (near, far, top, bottom, right, left)

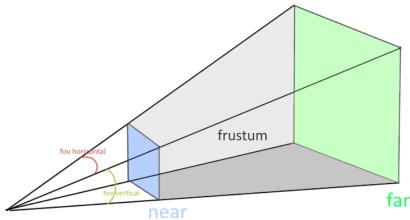


Figura 2.13: Diagrama del frustum de la cámara, siendo la zona gris el espacio de renderizado

Estos seis planos se codifican en la matriz de proyección, todo vértice fuera de este espacio durante la transformación quedarán descartados, esto se hace comparando las tres componentes x,y,z con la componente de coordenada homogéneas, si alguno de ellos es mayor o menor a w, ese vértice queda descartado.

La forma y la extensión del frustum determina el tipo de proyección del espacio 3D a la pantalla 2D, si los planos far y near tienen la misma dimensión, el frustum tendrá una proyección ortogonal, en otro caso, será una proyección en perspectiva.

Además determina el cambio de visión o field of view (FOV) que establece cuánto del espacio 3D será capturado por la cámara, este valor es dado en grados.

### Transformación de ventana o ViewPort

La multiplicación de las matrices anteriores tienen como resultado en la transformación de los vértices del modelo a un espacio coordenadas denominado clip, el cual es el espacio de coordenadas a renderizar en pantalla:

$$\text{VerticesClip} = \text{MatrixProyección} * \text{MatrizVista} * \text{MatrizModelo} * \text{VerticesModelo}.$$

Este resultado ha de ser asignado a `gl_Position` en el Vertex Shader para que WebGL realice las operaciones de representar las coordenadas 3D a coor-

nadas 2D, lo que se conoce en inglés como normalized-device coordinates, y mediante la función ViewPort transforme esta representación al espacio de coordenadas correspondiente al elemento HTML5 canvas (conocido como coordenadas de pantalla).

### Transformaciones de la normal

Otro elemento que debe de ser transformado son las normales de los vértices para que apunten en la dirección correcta, si aplicamos las mismas transformaciones que los vértices no siempre se mantiene la perpendicularidad de las normales.

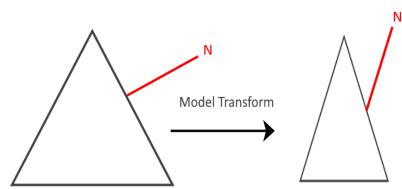


Figura 2.14: Ejemplo del uso erróneo de la matriz de transformación sobre la normal

Para calcular de forma correcta las normales se ha de encontrar una nueva transformación que mantenga la perpendicularidad, es decir:

$$N' \cdot S' = 0$$

Si llamamos  $M$  la transformación de la matriz de modelo-vista y  $K$  la nueva matriz a encontrar, sustituyendo  $N'$  y  $S'$ , obtenemos

$$(KN) \cdot (MS) = 0$$

El producto cruz se puede escribir como la multiplicación vectorial con el primer vector traspuesto:

$$(KN)^T (MS) = 0$$

Haciendo transformaciones obtenemos que:

$$N^T (K^T M) S = 0$$

Teniendo en cuenta que

$$N \cdot S = 0$$

entonces

$$N^T S = 0$$

, por lo tanto para cumplir con la ecuación anterior

$$K^T M = I$$

debe ser la matriz identidad, I

$$K^T M = I$$

Y aplicando álgebra se obtiene que:

$$K = (M^{-1})^T$$

Por lo tanto la matriz que necesitamos se obtiene de trasponer la inversa de la matriz modelo-vista.

### Implementación de la cámara

Tras tener en cuenta todo lo anterior, vemos que las matrices necesarias para implementar la cámara son:

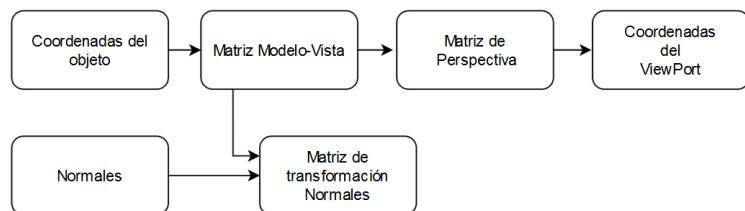


Figura 2.15: Diagrama donde se ve el flujo de matrices necesarias para generar la escena

- La matriz de modelo-vista: que al multiplicarla por los vértices obtenemos las coordenadas de vista.
- La matriz normal: Necesaria para que las normales sean perpendiculares a la superficie.
- La matriz de perspectiva.

### Transformaciones de la cámara

La matriz de modelo-vista permite el movimiento de la cámara por el escenario, la codificación de las transformaciones son:

$$\begin{array}{cccc}
 \text{Eje X} & \text{Eje Y} & \text{Eje Z} & \text{Traslación} \\
 \boxed{m1} & \boxed{m5} & \boxed{m9} & \boxed{m13} \\
 \boxed{m2} & \boxed{m6} & \boxed{m10} & \boxed{m14} \\
 \boxed{m3} & \boxed{m7} & \boxed{m11} & \boxed{m15} \\
 \boxed{m4} & \boxed{m8} & \boxed{m12} & \boxed{m16}
 \end{array}$$

Figura 2.16: Matriz que define la cámara

Donde las rotaciones quedan determinadas por las tres primeras filas y columnas, donde al inicializarse quedan como:

$$[m1, m2, m3] = [1, 0, 0]$$

$$[m5, m6, m7] = [0, 1, 0]$$

$$[m9, m10, m11] = [0, 0, 1]$$

Las traslación queda codificada en la última columna, sus tres primeras componentes, donde al inicializar queda:

$$[m13, m14, m15] = [0, 0, 0]$$

La última fila no tiene ningún significado especial, donde m4, m8, m12 son siempre 0 y m16 que es la coordenada homogénea siempre está a 1

Por lo tanto en la primera inicialización de la matriz de modelo-vista es igual a la matriz identidad de rango cuatro.

Hay que tener en cuenta que la cámara no existe como tal, si no que desde el origen de la escena vemos hacia la coordenada negativa de la z, y lo que realmente movemos son los modelos renderizados

Significa que todos los movimientos de la cámara deben ser opuestos, si se desea mover la cámara hacia adelante, se debe mover hacia atrás los modelos de la escena, si se desea rotar la cámara hacia la izquierda, los modelos de la escena deben rotar a la derecha.

Por lo tanto, los movimientos de la cámara se han de reflejar como la inversa de la matriz modelo-vista, esta matriz inversa nos permite trabajar en coordenadas de la cámara.

### Tipos básicos de cámaras

Existen dos tipos de cámaras que se puede implementar:

- La cámara orbital, usada en juegos de tercera persona, los giros en esta cámara se hacen en relación al centro del mundo, y las traslaciones alejándose se acercándose al centro.
- La cámara de seguimiento, usada en los juegos de primera persona, donde se puede mirar alrededor girando la cámara y movernos en la orientación donde apunta la cámara.

Es importante tener en consideración el orden en el que aplicamos las multiplicaciones difiere el resultado, no es lo mismo rotar sobre un origen y después trasladarlo, que en este caso sería la cámara orbital, que trasladar el origen y rotar, que en este caso sería la cámara de seguimiento.

$$RT \neq TR$$

#### 2.1.4. Texturas

Esta sección aborda los pasos necesarios para lograr añadir imágenes a los modelos 3D de la escena renderizada, siguiendo para ello la explicación proporcionada por los autores Ghayour y Cantor [11].

Las texturas son imágenes que se añaden a la superficie de la geometría renderizada, para que WebGL sepa que parte de la textura va en la superficie se necesita de otro atributo para el Vertex Shader conocido como coordenadas de texturas.

Las coordenadas de texturas es un conjunto de coordenadas de dos dimensiones de tipo flotante, cuyos rango de valores van del 0 al 1, siendo la coordenada (0,0) la esquina izquierda superior y la coordenada (1,1) las esquina derecha inferior.

En OpenGL y en WebGL a las coordenadas de textura se hace referencia como (s,t) para la coordenada x e y respectivamente.

El proceso de añadir texturas es muy similar al proceso de crear y añadir datos al buffer de vértices.

Primero se debe crear una textura:

```
1 const texture = gl.createTexture();
```

Después hay que enlazarlo para que WebGL pueda manipularlo:

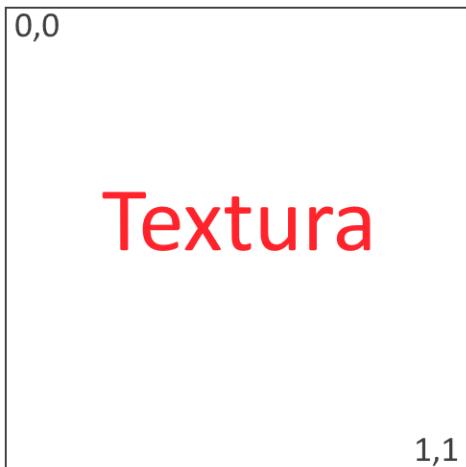


Figura 2.17: Ejemplo de una textura donde se indica las coordenadas mínimas (0,0) y máximas (1,1) para referenciar un pixel en este

```
1 gl.bindTexture(gl.TEXTURE_2D, texture);
```

A continuación se le provee con los datos de la imagen:

```
1 gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,  
    imagen);
```

Los datos de la imagen pueden ser provistos directamente desde el DOM (`document.getElementById('texture-image')`) o mediante un objeto JavaScript Image. Los tipos de datos soportados son los mismos que soporta el navegador web.

Las dimensión de la imagen se determina de forma automática por lo que no es necesario indicarlo.

Posteriormente se le indica WebGL que genere una colección de Mipmap, que son un conjunto de texturas gradualmente disminuidas, que se generan por interpolación bilineal.

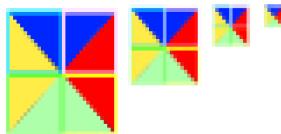


Figura 2.18: imagen de ejemplo del uso de Mipmap

Esto permite acelerar el proceso de calcular el color a pintar en tiempo de

renderizado cuando la superficie es más pequeña que la textura.

Y por último se le indica a WebGL los parámetros de filtrado que se aplicarán en la textura cuando se renderizan, por lo general no siempre la textura tiene un ajuste 1:1 con los pixeles de la pantalla, si no que se ha de ajustar para mostrarse, por ejemplo cuando la superficie a aplicar la textura está lejos de la cámara y no se puede mostrar la textura entera, tal como ocurre en la imagen de abajo:

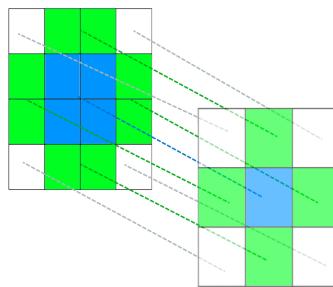


Figura 2.19: Ejemplo del proceso de rasterización donde no es 1:1 la conversión.

La función que nos permite determinar esto es:

```
1 texParameterf(target, pname, param);
```

Donde:

1. *Target* especifica el tipo de textura, puede ser

- *gl.TEXTURE\_2D* : Una textura de dos dimensiones.
- *gl.TEXTURE\_CUBE\_MAP*: Imagen mapeada para un cubo.
- *gl.TEXTURE\_3D* : Una textura tridimensional.
- *gl.TEXTURE\_2D\_ARRAY*: un array de textura bidimensional.

2. *Pname* indica el parámetro a usar, existen varios, entre los que destacan:

- *gl.TEXTURE\_MAG\_FILTER*: Cuando el muestreo de la textura determina que la textura debe aumentarse, se le especifica cómo debe de calcular el color, acepta los valores:
  - *GL\_LINEAR*: Devuelve la media ponderada del color de textura que estén más cerca de las coordenadas de textura especificadas. (valor por defecto)

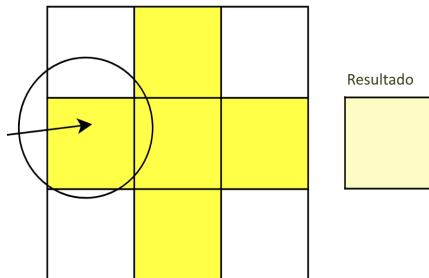


Figura 2.20: imagen de un supuesto uso del filtro linear

- *GL\_NEAREST*: Devuelve el valor del color de textura más cercano (en la distancia de Manhattan) a las coordenadas de textura especificadas.

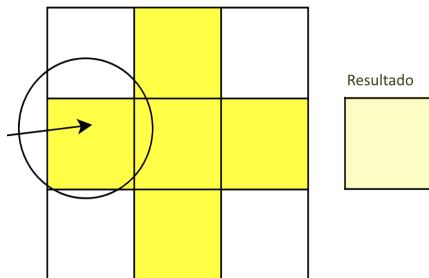


Figura 2.21: imagen de un supuesto uso del filtro nearest

- *gl.TEXTURE\_MIN\_FILTER*: Cuando el muestreo de la textura determina que la textura debe aumentarse, se le especifica cómo debe de calcular el color, acepta los valores:
  - *GL\_LINEAR*: Devuelve la media ponderada de los colores de la textura que estén más cerca de las ubicación especificadas
  - *GL\_NEAREST*: Devuelve el color de la textura más a las coordenadas especificadas mediante el cálculo de la distancia de Manhattan.
  - *NEAREST\_MIPMAP\_NEAREST* : Elige el mipmap de tamaño más cercano y elige un pixel de este mipmap. (valor por defecto)
  - *LINEAR\_MIPMAP\_NEAREST*: Elige el mipmap de tamaño más cercano hace un promedio del color de las cuatro texturas más cercanas a la coordenada especificada.

- *NEAREST\_MIPMAP\_LINEAR*: Elige dos mipmap que más se ajustan al tamaño y elige el color de un pixel de ambos, y pondera el color.
- *LINEAR\_MIPMAP\_LINEAR*: Elige dos mipmap que más se ajustan al tamaño y elige el color de cuatro pixeles de ambos, y pondera el color.
- *g.TEXTURE\_WRAP\_S* Y *gl.TEXTURE\_WRAP\_T*: Indican que debe se debe de hacer cuando se le indica una de coordenada de textura fuera del rango en el eje x e y respectivamente, los valores que acepta son:
  - *gl.REPEAT*: Repite la textura. Comportamiento por defecto.

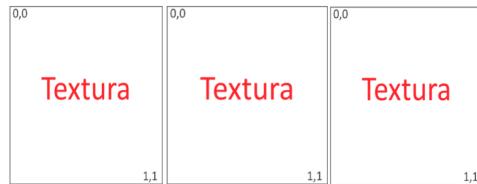


Figura 2.22: imagen de un supuesto uso del filtro repeat

- *gl.CLAMP\_TO\_EDGE*: Repite solo los bordes de la textura.
- *gl.MIRRORED\_REPEAT*: Repite la textura pero en cada repetición la muestra girada como en el reflejo de un espejo.



Figura 2.23: imagen de ejemplo del uso del filtro MirrorRepeat

Para usar la textura en el Shader se necesitan de un atributo de entrada donde indicar las coordenadas de texturas de tipo `vec2` y una variable uniforme para acceder a los datos de la textura de tipo `sampler2D`. Por último se le indica en el fragment shader se le indica al fragment color que use la textura:

```
1 fragColor = texture(textureData, textureCoords);
```

# Capítulo 3

## Desarrollo del TFM

### 3.1. Tecnologías del proyecto

#### OpenGL

Antes de realizar la introducción a WebGL es conveniente hacer una mención a la librería sobre la cual WebGL se basa, esta es OpenGL. De Vries [13] expone en su libro que OpenGL es una especificación que describe cómo las aplicaciones deben interactuar con los componentes de hardware de la tarjeta gráfica para producir imágenes en la pantalla. Se considera principalmente una API, es decir es una especificación que indica qué tipo de resultados o entradas tendrá cada función y cómo debería operar. Las librerías de OpenGL son implementaciones de la especificación que normalmente son realizadas por los fabricantes de tarjetas gráficas. Actualmente, la especificación de OpenGL está mantenida y desarrollada por el grupo Khronos.

#### WebGL

Atendiendo a la definición dada por Matsuda y Lea [14], WebGL es una tecnología que permite dibujar, mostrar por pantalla e interaccionar con elementos 3D desde el navegador web. Aunque la programación gráfica ha estado tradicionalmente restringida a ordenadores o consolas de videojuegos, los navegadores web modernos permiten crear aplicaciones gráficas 3D mediante tecnologías web que son accesibles y bien conocidas. Por lo tanto, WebGL junto con HTML5 y JavaScript permiten a los desarrolladores crear aplicaciones 3D accesibles desde el navegador web, haciéndolas altamente portables y soportadas por multitud de dispositivos, como smartphones, tablets, etc.

## HTML5

HTML5 es la última evolución del estándar HTML que mejora y amplía sus características, entre ellas, la capacidad de manejar gráficos 2D, servicios de red y acceso a almacenamiento local. Esto le ha permitido evolucionar de un simple motor de presentación a una sofisticada plataforma de aplicaciones. Junto con WebGL, HTML5 añade una nueva capa para la creación de aplicaciones y experiencias 3D basadas en el navegador.

Antes crear una aplicación 3D implicaba realizar un programa para un tipo de dispositivo, usando lenguajes como C o C++ en conjunto a librerías gráficas como OpenGL o Direct3D. Sin embargo, ahora, WebGL permite un desarrollo orientado al navegador usando HTML y JavaScript, por lo que no es necesario instalar ningún tipo de librería o plug-in extra.

## JavaScript

Siguiendo a Rauschmayer, [15] JavaScript es un lenguaje de programación que se introdujo en 1995 con el fin de añadir funcionalidad a las páginas web en el navegador Netscape. Aunque se considera que ECMAScript y JavaScript son lo mismo, en realidad ECMAScript es una especificación que describe cómo los navegadores web deben implementar el lenguaje JavaScript. Los estándares ECMA-262 y el ISO/IEC 1626 son las principales especificaciones de ECMAScript y JavaScript, respectivamente.

Con el tiempo el estándar ha ido evolucionando como vemos en la siguiente tabla 3.1:

Tabla 3.1: Tabla donde se muestra la evolución del estandar ECMAScript con sus principales mejoras

Versión	Fecha implantación	Cambios
ECMAScript 1	Junio de 1997	Primera versión del estándar.
ECMAScript 2	Junio de 1998	Actualizaciones pequeñas para sincronizar los dos estándares ECMA-262 y ISO.
ECMAScript 3	Diciembre de 1999	Expresiones regulares, estructuras de control como do-while, switch, manejo de excepciones con try/catch.
ECMAScript 4	Abandonado en Julio de 2008	Añade grandes actualizaciones pero tuvo muchas controversias, se decido abandonar la versión.
ECMAScript 5	Diciembre 2009	Se añade pequeñas mejoras a la versión 3.
ECMAScript 6	Junio 2015	Añade las mejoras que se esperaban desde la versión 4, como módulos, espacio de nombres, etc...

El estándar ECMAScript ha sufrido actualizaciones prácticamente todos los años, desde la versión ECMAScript de 2016 a cada versión se le referencia con el año de salida. Actualmente, la versión del estándar es ECMAScript 2023.

JavaScript es un lenguaje de programación que ha sido adaptado a la gran mayoría de los navegadores web, lo que hace posible la creación de aplicaciones web en las que el usuario puede interactuar con la página sin necesidad de recargar en cada acción que quiera llevar a cabo. Además, JavaScript presenta una sintaxis flexible en comparación con lenguajes de programación convencionales como C o Java. Por ejemplo, no requiere declaración de tipo en las variables, lo que permite asignación dinámica. En caso de error, no para la ejecución del programa, lo que facilita la depuración. Según el autor, esta flexibilidad presenta ventajas, como la facilidad de aprendizaje para programadores nuevos, pero, a cambio, hace que sea difícil manejar errores.

Entre las principales características de JavaScript según en el sitio oficial de Mozilla [16] se pueden citar:

- Es un lenguaje de programación interpretado que se ejecuta en el lado del cliente, aunque también puede llegar a ser ejecutado en el lado servidor mediante el entorno NodeJS.
- El tipo de las variables es en tiempo de ejecución, por lo que se considera dinámico.
- Es muy usado en conjunto con HTML y CSS para el desarrollo de páginas web dinámicas, permitiendo cosas como mapas interactivos, animaciones gráficas, etc.
- Permite el uso de APIs en el lado cliente, como las APIs de Canvas y WebGL que son el objeto de estudio de este trabajo fin de máster, los cuales permiten crear gráficos 2D y 3D.

## TypeScript

Según la información proporcionada en su sitio web oficial [17] TypeScript es un lenguaje de programación que se desarrolló con la idea de hacer de JavaScript un lenguaje idóneo para proyectos de gran envergadura añadiendo características extras como tipado estático y comprobación de errores en el tiempo de compilación. TypeScript es un superconjunto de JavaScript, por lo que comparten sintaxis. Compilar código TypeScript generará código JavaScript, lo que mantendrá todas las ventajas de portabilidad añadiendo las características anteriormente indicadas.

A modo de ejemplo, si ejecutamos el siguiente código JavaScript:

```
1 | console.log(4 / []);
```

Se muestra en el log del navegador 'Infinity', no avisando de ningún error, sin embargo, esto no estaría permitido por TypeScript, que mostraría un mensaje de error, indicando el tipo de dato que se esperaba en el lado derecho de la operación.

Por lo expuesto se ha decidido hacer uso de TypeScript, pues nos ofrece una sintaxis sencilla y un control en los tipos de datos, permitiendo detectar y controlar errores de forma más sencilla que si se hubiera elegido JavaScript.

## 3.2. Capa software para aplicaciones WebGL

Para facilitar el desarrollo del juego se realiza una capa software que reúne todas las funcionalidades WebGL necesarias para el renderizado, evitando mezclar en el código partes propias de WebGL con las del juego, logrando así un diseño con un bajo acoplamiento y una alta cohesión.

Por ello, todos los elementos descritos anteriormente en la sección 2.1 conforman un desarrollo propio diferente a la del juego, donde a través de la clase *WebGLController* se ofrece una interfaz para acceder a toda las funcionalidades necesarias para renderizar, siendo estas accedidas en la implementación del juego.

A continuación se describen las clases resultantes del desarrollo realizado para la implementación de las secciones anteriormente tratadas de WebGL.

### Clases TypeScript

Las clases realizadas en el proyecto de WebGL son:

- *WebGLController*: Clase principal que da acceso a toda la funcionalidad necesaria para poder cargar y renderizar la escena 3D, encargada de llamar a los métodos de las otras clases para ponerlas a disposición en un único lugar a modo de interfaz, logrando diferenciar la implementación de WebGL con respecto al del juego. Entre sus métodos más importantes:
  - *drawModel(model: String, matrixModel : array, animation : int, frame : int) : void*, encargada activar el VAO del modelo a pintar, indicando para ello, la matriz de transformación del modelo y que animación en que frame se pintará.

- *Models*: Clase que contiene un array asociativo que relaciona el nombre del modelo con su objeto modelo, facilitando su acceso, además, controla que los modelos ya inicializados no vuelvan a ser cargados. Entre sus métodos más importantes:
  - *getVAO(animacion : int, frame : int) : VAO*, Devuelve el VAO asociado a un modelo para una de sus animaciones en un frame concreto.
- *Model*: Clase que representa un modelo, contiene el conjunto de VAOs que conforman las animaciones, la imagen de textura y los atributos del material. Entre sus métodos más importantes:
  - *cargarVAOdeOBJ(ruta: string, nombreModelo: String) : void*, crea un VAO a partir de los datos de un fichero OBJ indicado.
- *VAO*: Clase que se encarga de crear los buffers necesarios, añadir los datos y asociarlos a un VAO.
- *LectorOBJ*: Se encarga de la lectura de los ficheros OBJ y de parsearlos a un objeto JSON facilitando así el posterior acceso a los datos del OBJ.
- *Camara*: Clase que modela la cámara descrita en los apartados anteriores, en ella están la matriz de proyección y la matriz de la cámara y es la encargada de controlar todas las operaciones que se realizan con la cámara (traslaciones, rotaciones, ..)
- *Lights*: Esta clase modela las luces puntuales que existen en el juego, especificando su posición y sus propiedades ambiental, difusa, especular.
- *ProgramShader*: Clase encarga de compilar los Shader y vincular sus atributos y uniformes para ser accesibles desde el programa.
- *VertexShader y FragmentShader*: Estas clases contienen el código GLSL de los shaders implementados.

## Diagrama de clases

El diagrama de clase que conforma este desarrollo es el siguiente:

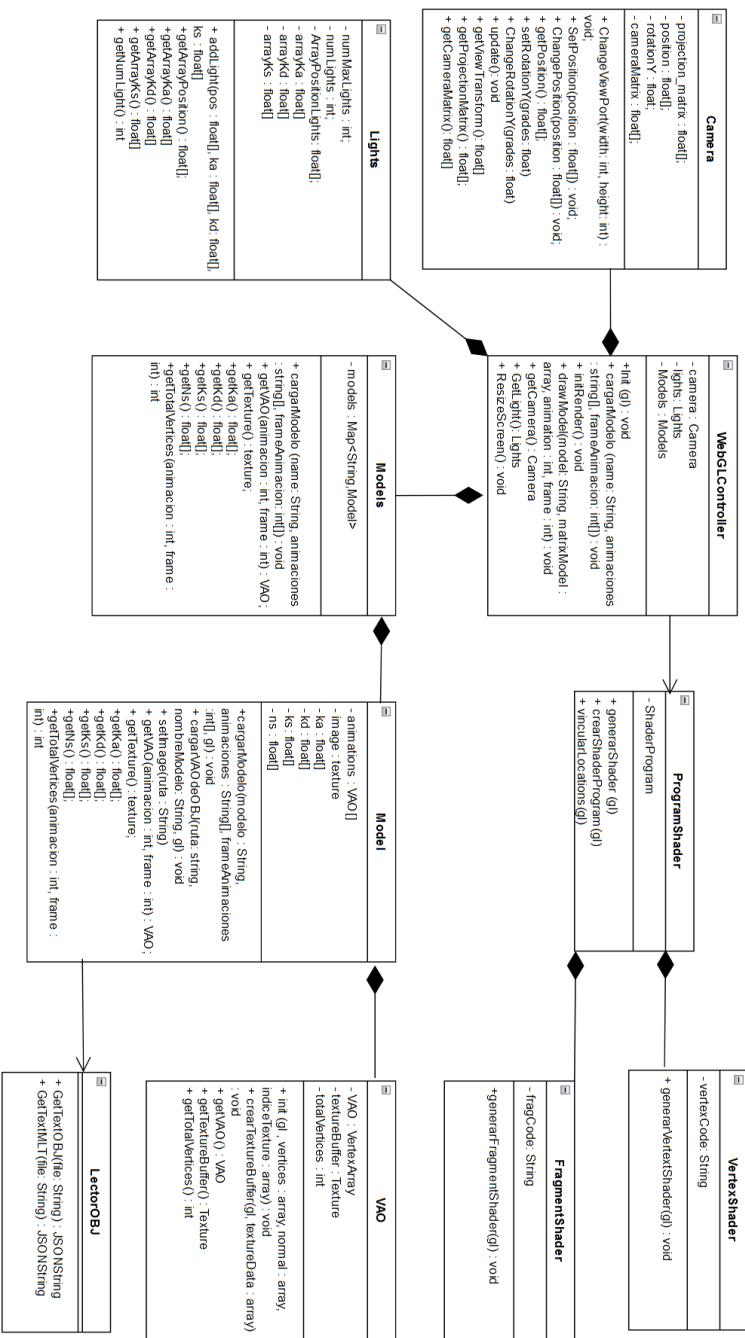


Figura 3.1: Diagrama de clases del proyecto de WebGL

### 3.3. Análisis y Diseño del software de Videojuego

El juego desarrollado para este trabajo fin de máster consiste en un juego de exploración de mazmorras o dungeon crawler con elementos RPG (role-playing game) con un diseño fantasía medieval donde se tomará el control de un aventurero que debe explorar mazmorras en busca de tesoros, haciendo frente a los enemigos que habitan en ella.

#### 3.3.1. Requisitos funcionales

Los elementos de dungeon crawler que se incorporan en el juego son:

- Escenarios con forma de mazmorras laberínticas, siendo estas generadas aleatoriamente.
- Conjuntos de mazmorras consecutivas que se han de superar para ganar el juego.
- Mapeado de las mazmorras en forma de casillas.
- Objetos esparcidos por las mazmorras para ayudar el jugador en su aventura, por ejemplo pociones.
- Enemigos que atacarán al personaje durante su exploración.
- Existencias de mapas que nos permitirán orientarnos en la mazmorra.

Los elementos de RPG que se usan en este juego son:

- Nivel de personaje, que hace referencia a que tan poderoso es.
- Puntos de experiencia, que se consigue ganando combates y permiten subir de nivel.
- Puntos de vida, número que indica la salud del personaje, se reducirá al recibir daño y se aumentará al curarse.
- Puntos de daño, representa el daño que se ocasiona sobre los puntos de vida.
- Turnos de acción, que controla el orden en el cual los personajes realizan sus acciones.

Para hacer frente a los enemigos el protagonista contará con dos armas, una espada y un arco. La espada permite atacar a los enemigos adyacentes, y el arco a los enemigos cercanos hasta cierta distancia.

El objetivo principal del juego consiste en sobrevivir hasta alcanzar la décima mazmorra, que es donde se encuentra la sala del tesoro.

### **3.3.2. Lógica del juego**

La lógica del juego determina la situación actual en la que se encuentra la partida, las acciones posibles que se pueden llevar a cabo y en qué orden se han de realizar.

Al iniciar la partida se realiza:

- La creación del mapa.
- La creación del protagonista, enemigos y elementos restantes del juego: suelos, paredes, techos, pociones y flechas.
- Colocar los elementos anteriores en una posición aleatoria del mapa.
- Crear un objeto turnos y darle los identificadores del protagonista y los enemigos, este objeto crea una cola, donde el primer elemento representa al personaje que le toca realizar alguna acción.

Todos estos pasos anteriores se agrupan en un método llamado initScene(), permitiendo su reutilización al ir cambiando de mazmorras.

Posteriormente se pasa a la fase de juego, donde el personaje al que le toque su turno realiza una acción, las acciones posibles son:

- Moverse a una casilla adyacente.
- Atacar.
- Usar una poción, esto solamente el protagonista.
- Coger flechas y pociones, esto solamente el protagonista.
- Pasar a la siguiente mazmorra.

Si se realiza la acción de moverse, se procede el cambio de posición a una casilla adyacente, para ello se realizan las siguientes acciones:

1. Se comprueba que la casilla adyacente no sea pared y no esté ocupada, si no es así finaliza la acción de moverse.
2. Se calcula el desplazamiento desde la casilla ocupada actualmente a la casilla adyacente a la que se dirige y se realiza la animación de desplazamiento.

El cambio de casilla sólo está permitido en una de las cuatro casillas adyacentes (arriba, abajo, izquierda y derecha), Cuando llegue la casilla objetivo la acción de desplazamiento acaba.

La acción de atacar conlleva la realización de las siguientes acciones:

1. Se comprueba si se daño a algún personaje (en caso de atacar el protagonista, se comprueba si se daño a algún enemigo, y viceversa), comprobando si está dentro del rango ataque, el ataque con espada tiene rango una casilla, el arco tres casillas, todos los enemigos tienen rango una casilla.
2. Se realiza la animación de ataque y la de recibir daño, esta última solamente si se daño a algún personaje.
3. En caso de dañar a algún personaje se le resta el daño a sus puntos de vida, y en caso de tener puntos de vida a cero, se elimina al enemigo o se pierde la partida si es el protagonista.
4. Si se elimina a un enemigo se suma a los puntos de experiencia del protagonista los puntos de experiencia que se dan por derrotar a ese enemigo concreto y se comprueba si subió de nivel.

Para la acción de usar pociones primero se comprueba si hay existencias, en caso positivo se aumenta la salud del protagonista, nunca por encima de su vida máxima.

La acción de coger flechas y pociones se hace automáticamente cuando se acaba la acción de moverse a otra casilla, es decir, para coger objetos hay que colocarse encima de este, por lo que cada vez que el protagonista se mueve a otra casilla, se comprueba si hay un objeto, en caso positivo, se comprueba que puede llevar más objetos de ese tipo (para pociones máximo diez y para flechas máximo veinte) si es así se aumenta en uno la cantidad de ese objeto en el protagonista y se elimina de la escena.

La acción de pasar a la siguiente mazmorra, se lleva a cabo cuando el protagonista se mueve a la casilla donde está la escalera que da acceso a la siguiente mazmorra, esta acción llama al método initScene() donde se generará una nueva mazmorra y enemigos, Pero si la mazmorra siguiente es la décima entonces se pasa al estado de partida ganada.

### 3.3.3. Generación de las mazmorras

La mazmorra consiste en un conjunto de habitaciones unidas por pasillos.

Para la generación de las mazmorras se hace uso de una matriz cuadrada de dimensión 50, donde cada celda de la matriz representa una casilla en la mazmorra. Las casillas de la mazmorra pueden ser:

- Habitaciones y Pasillos, espacios donde los personajes pueden desplazarse, se representan en la matriz con un 1.

- Paredes, espacios que no pueden ser ocupados ni desplazarse sobre ellos, se representan en la matriz con un 0.
- Casillas ocupadas, espacios que ya están siendo ocupados por algún personaje, por lo que ningún otro lo podrá ocupar, se representan con un 2 en la matriz.

Para la generar la mazmorra de forma aleatoria se va hacer uso de la técnica de la partición del espacio binario, en el cual de forma recursiva de subdividir un espacio en un conjunto de dos, donde la condición de parada es que la subdivisión realizada tenga un tamaño entre un máximo y un mínimo.

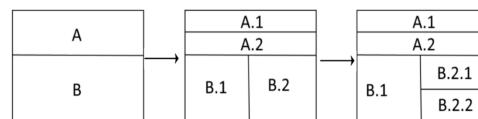


Figura 3.2: Ejemplo del proceso de la técnica de la partición del espacio binario.

Una vez realizada las subdivisiones se coge el punto central de cada una y crea una habitación dentro de los límites de la subdivisión, de tamaño aleatorio, siendo estas posteriormente unidas por los pasillos, tal y como se muestra en la figura 3.3:

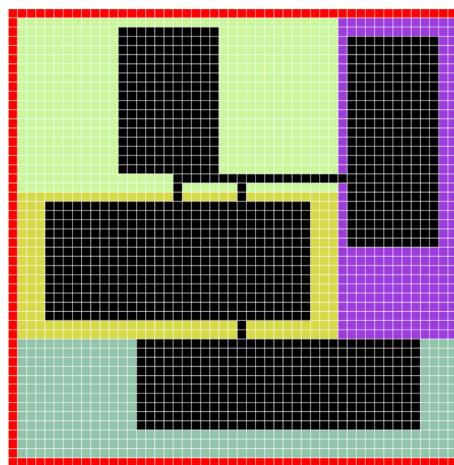


Figura 3.3: Ejemplo de un mapa aleatorio usando la técnica de la partición del espacio binario.

Donde se aprecian las subdivisiones realizadas en los distintos colores, y en negro las habitaciones y los pasillos.

A continuación se expondrá de forma resumida el algoritmo implementado en pseudocódigo:

```

1 Funci n dividir(desdeX : entero, desdeY : entero, hastaX : entero,
      hastaY : entero){
2     si hastaY - desdeY y hastaX - desdeX >
        tama oM ximoHabitaci n entonces:
3         random : numero aleatorio entre 0 y 1
4         si random es 0: //se hace un corte horizontal
            //se calculan las nuevas coordenadas para un corte
            //horizontal
5             nuevaHastaY = (desdeY + hastaY) / 2; //punto de
            //corte
6             nuevaHastaX = hastaX;
7             nuevadesdeY = nuevaHastaY;
8             nuevadesdeX = desdeX;
9         sino:
10            //se calculan las nuevas coordenadas para un corte
            //vertical
11            nuevaHastaY = desdeY;
12            nuevaHastaX = (desdeX + hastaX) / 2; //punto de corte
13            nuevadesdeY = desdeY;
14            nuevadesdeX = nuevaHastaX;
15        fin si
16        dividir(desdeX, desdeY, nuevaHastaX, nuevaHastaY); //primera
            mitad
17        dividir(nuevaDesdeX, nuevaDesdeY, hastaX, hastaY); //segunda
            mitad
18        sino: // si no puedo seguir dividiendo genero la habitaci n
            crearHabitaci n(desdeX, desdeY, hastaX, hastaY);
19        fin si
20    }
21 }
22 }
```

Una vez hechas las divisiones, por cada división se crea un cuadrado dentro del espacio de tamaño aleatorio que representa la habitación y por último pasando de habitación en habitación se unen mediante pasillos.

### 3.3.4. Desarrollo software del juego

En esta sección se describe el desarrollo de clases realizada para la implementación de las secciones anteriormente tratadas sobre el juego.

Indicar que las clases detalladas a continuación hacen uso de la interfaz ofrecida por la clase *WebGLController* en la capa software explicada en la sección 3.2, para acceder a toda las funcionalidades de renderizado.

#### Clases TypeScript

Las clases realizadas en el proyecto del juego son:

- *Juego*: Clase principal encargada del funcionamiento del juego, desde cargar las pantallas, inicializar los modelos, controlar la lógica del jue-

go e indicar que es lo que se debe renderizar en cada momento. Sus principales métodos:

- *init()*: Encargada de la inicialización del controlador de WebGL.
  - *initScene()*: Crea los objetos del juego y le indica a webGL que cargue los modelos.
  - *logic()*: Método que controla la ejecución de las acciones por parte de los personajes del juego mediante un objeto de la clase Turno. Representa el bucle principal del juego.
  - *render()*: Le indica a webGLController y a los objetos de Interface y Pantallas qué es lo que se debe de pintar en cada momento según la situación en la que se encuentra la partida, esto se controla mediante una variable llamada estadoDelJuego.
  - *accionesProtagonistas()*: Encargada de llevar a cabo las acciones que el protagonista puede realizar.
  - *accionesEnemigo()* : Método que lleva a cabo las acciones de los enemigos.
  - *atacar()* : Controla la secuencia de eventos del ataque.
- *Modelo*: Clase encargada de modelar un objeto 3D del escenario y controlar su posición y rotación en la escena.
  - *Protagonista y Enemigos* : Clases que heredan de Modelo, y tiene los atributos propios RPG de los personajes, como los puntos de vida, el nivel, etc.. Se diferencian en que Protagonista tiene acciones como usar o coger flechas y pociones.
  - *Turnos*: Clase encargada almacenar en una cola los ids de los personajes para controlar el orden de acción.
  - *Reloj*: Encargada de controlar el paso del tiempo en milisegundos o DeltaTime, imprescindible para controlar por ejemplo las animaciones.
  - *Mapa*: Clase que dirige el proceso de creación del mapa según se explicó en el apartado de generación de mapa.
  - *Interface*: Clase que representa la interfaz del usuario, entre sus funciones, mostrar la barra de vida, el número de pociones y flechas disponibles, mostrar las 5 últimas entradas del log, mostrar el minimapa, mostrar el mapa. Además, como se tratará en el capítulo de portabilidad, también se encarga de verificar si es un dispositivo móvil para mostrar una interfaz u otra.
  - *Pantallas*: Se encarga del dibujado de las pantallas por la que pasa la partida, como la pantalla de inicio, la pantalla de carga, etc..

**Diagrama de clases**

El diagrama de clase que conforma este desarrollo es el siguiente:

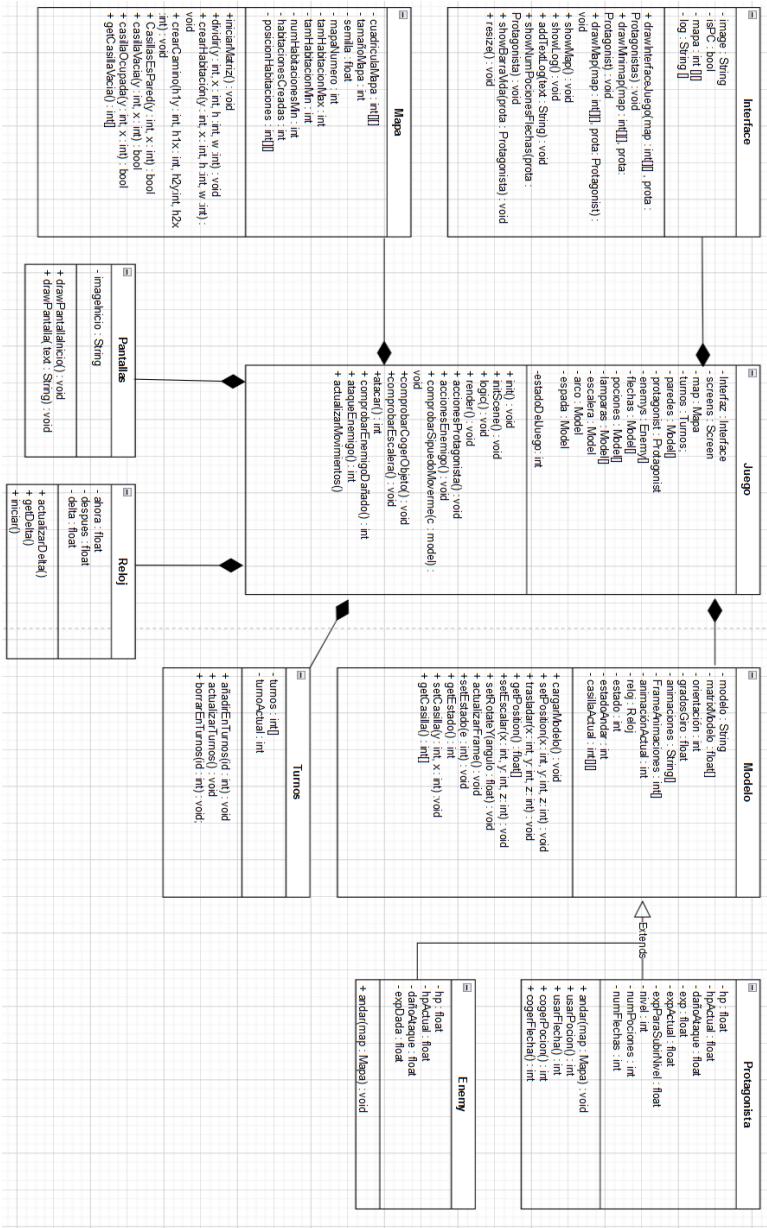


Figura 3.4: Diagrama de clases del juego

### 3.4. Diseño visual y modelado

Para el diseño del juego se ha elegido una ambientación medieval oscura, siendo esta la temática más usada en el género de los dungeon crawlers, donde las mazmorras están hechas piedra, los enemigos son criaturas fantásticas

como ogros, hay poca iluminación ambiental, etc..

Para la creación de los modelos 3D se ha usado el software Blender y para los elementos 2D, como la interfaz del usuario, el software de edición de imágenes Paint.NET.

En los siguientes apartados se tratará el diseño elegido para cada elemento del juego.

### Diseño de los modelos de personajes y elementos del juego

Existen dos tipos de enemigos en el juego:

1. Ogros: Los cuales se suelen representar como seres humanoides grandes con un color de piel verdoso.



Figura 3.5: Modelo 3D del ogro

Donde sus animaciones son tres: andar, atacar y recibir daño.

2. Murciélagos: Criaturas que suelen habitar en las mazmorras.



Figura 3.6: Modelo 3D del murciélagos

Donde sus animaciones son tres: andar, atacar y recibir daño.



Figura 3.7: Ejemplos de frames del ogro en la animación de andar



Figura 3.8: Ejemplos de frames del ogro en la animación de atacar



Figura 3.9: Ejemplos de frames del ogro en la animación de recibir daño



Figura 3.10: Ejemplos de frames del murciélagos en la animación de volar.



Figura 3.11: Ejemplos de frames del murciélagos en la animación de atacar.

A continuación se mostrarán los elementos que conforman la mazmorra:

- a) Muros: Compuesto de piedras, en cada cara se muestra un patrón distinto para que no sea tan repetitivo al colocarlo en la mazmorra.



Figura 3.12: Modelo 3D del muro

b) Suelo: También de piedra, con un diseño de bloques.



Figura 3.13: Modelo 3D del suelo

c) Techo: De madera, donde se aprecia una zona central saliente y más clara, que hace de viga que soporta el techo.

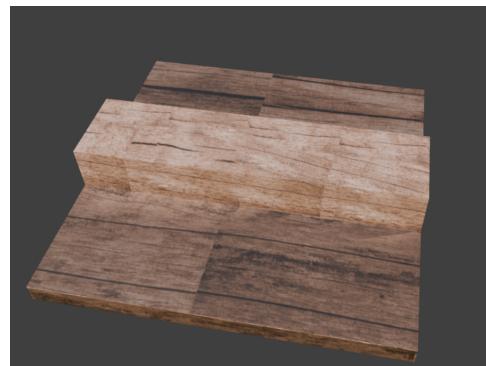


Figura 3.14: Modelo 3D del techo

El resultado de la unión de estos tres elementos es el siguiente:



Figura 3.15: Escena resultante al combinar modelos, texturas y luces

A continuación se muestran los elementos que se encuentran repartidos por el mapa:

- a) Pociones: Botella de cristal con un líquido rojo en su interior que cura la salud.



Figura 3.16: Modelo 3D de una poción

- b) Flechas: Munición del arco del personaje.



Figura 3.17: Modelo 3D de una flecha

- c) Escaleras: Elemento que permite avanzar entre mazmorras.



Figura 3.18: Modelo 3D de las escaleras

Debido que el juego es en primera persona, no existe un modelo para el protagonista, solamente se mostrará la espada y el arco al usarlas.

- a) Espada: Clásica espada de metal de doble filo.



Figura 3.19: Modelo 3D de la espada

- b) Arco: Clásico arco de madera para disparar las flechas.



Figura 3.20: Modelo 3D del arco

### Diseño de la interfaz de usuario

La interfaz del usuario permite al jugador tener la información más relevante de la partida en la pantalla, para este proyecto la información que se mostrará será:

- Barrada vida, la cual se llena o vacía según los puntos de vida totales y actuales del personaje, además de mostrarse también de forma numérica.
- Un registro o log de los eventos que van sucediendo en el juego, por ejemplo, si se ha cogido un objeto.
- Un minimapa donde se muestra la información de las casillas alrededor del protagonista.
- Una imagen de una poción y una flecha donde se muestra el número de pociones y fechas que tiene el jugador.



Figura 3.21: Interfaz destinada a dispositivos de sobremesa y portátiles

Se ha buscando una distribución que no estorbe la visualización de la escena, además de incorporar textos con tamaño de letra que faciliten la lectura.

En el apartado de portabilidad se habla de otra interfaz específica para dispositivos móviles debido a que requiere de otros tipos de elementos en pantalla, como por ejemplo la cruceta de movimiento.

### Diseño de las distintas pantallas del juego

Como es común en los juegos, estos están compuestos de distintas pantallas por el cual se irá navegando durante la partida, para este proyecto se ha utilizado las siguientes pantallas:

- a) Pantalla de inicio: Imagen en la que se muestra el título y el autor de este trabajo fin de máster.
- b) Pantalla de carga: Imagen que muestra el texto “Cargando”, la cual estará activa mientras se cargan los elementos del juego.

- c) Pantalla de juego: En donde se muestra el escenario renderizado 3D y la interfaz del usuario, aquí es donde se lleva a cabo la partida.
- d) Pantalla de derrota: Imagen que se muestra cuando el protagonista pierde todos sus puntos de vida.
- e) Pantalla de victoria: Imagen que se muestra cuando el protagonista llega a la planta objetivo.

La navegación entre las distintas pantallas se muestra en la siguiente imagen:

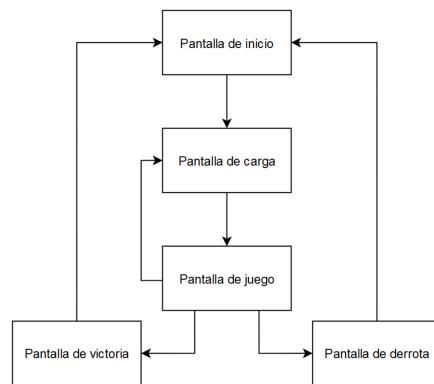


Figura 3.22: Diagrama de flujo de las distintas pantallas del juego donde se ve su navegación

Lo primero que se mostrará es la pantalla de inicio y hasta que no se pulse un botón cualquiera no cambia, una vez pulsado un botón, se pasa a la pantalla de carga la cual permanecerá activa hasta que no se haya finalizado la carga de los elementos del juego, finalizada la carga se pasa a la pantalla de juego en el que ocurre la partida, de aquí existen tres posibilidades, que se cambie de piso, por lo que vuelve a la pantalla de carga, o se gane o se pierda, llevando estos a su respectiva pantalla, desde las pantallas de victoria o derrota al pulsar cualquier botón se vuelve a la pantalla de inicio.

### 3.5. Portabilidad

Los dispositivos finales los cuales pueden ser objeto de estudio para este proyecto es muy amplio, debido a que la gran mayoría de dispositivos informáticos tienen instalado un navegador web. Este proyecto se centra en concreto en los siguientes dispositivos:

- Ordenadores de sobremesa y portátiles.
- Teléfono inteligente.
- Tablets.
- Consolas de videojuegos.

Todos estos ellos tienen unas características propias que hay que tener en cuenta a la hora del desarrollo del juego, principalmente:

- Características de entrada de los dispositivos.
- Adaptación de la interfaz a las características de los dispositivos.
- Rendimiento ofrecido.

### Características de entrada de los dispositivos

Este apartado hace referencia a las formas que ofrecen los dispositivos de los usuarios para poder controlar e interactuar con el juego. Dependiendo del tipo de dispositivo de entrada podemos encontrarnos con las siguientes opciones:

- Pantalla táctil, si el dispositivo es un teléfono inteligente o tablet.
- Teclado, para ordenadores de sobremesa y portátiles.
- Mando de juegos, puede ser tanto para consolas de videojuegos como para los dispositivos anteriormente nombrados.

JavaScript permite mediante el uso de eventos controlar las pulsaciones en cada uno de los dispositivos anteriormente mencionados, en concreto para este proyecto se usan los siguientes eventos:

- Para pantalla táctil:
  - Touchstart: El cual recoge en un objeto JavaScript evento que almacena la posición de pantalla pulsada. Se usará para comprobar que elemento de la interfaz se pulsó, además de añadir un pequeño sombreado en la zona donde se pulsó, a modo de feedback para dar entender que se pulsó correctamente.
  - Touchend: Se llama cuando se deja de pulsar la pantalla, se usará para eliminar el sombreado anteriormente descrito.
- Para teclado:
  - Keydown: El cual recoge en un objeto JavaScript evento que almacena el código ASCII de la tecla pulsada.
  - Keyup: Se llama cuando se deja de pulsar una tecla.

### Adaptación de la interfaz a las características de los dispositivos

Este apartado trata de cómo se han de mostrar los elementos del juego según el tamaño pantalla del dispositivo sobre el cual se esté ejecutando el juego, si se quiere conseguir un juego multiplataforma se debe dar una experiencia visual lo más similar posible en todos los dispositivos, para ello se tiene en consideración lo siguiente:

- Los elementos visuales se deben de adaptar al tamaño de la pantalla logrando que en todos se vean lo más parecido posible.
- Hay que analizar que elementos de la interfaz del juego se deben de mostrar y cuales no, para por ejemplo evitar que en pantallas pequeñas haya tanta información que resulte difícil su correcta visualización o su tamaño de letra sea difícil de visualizar.

### Adaptación al tamaño de pantalla

Dado que el juego se ejecuta sobre el navegador web, lo que interesa no es realmente el tamaño de pantalla, si no el tamaño de área de visualización en el navegador web, es por ello que hace uso del objeto de JavaScript "Windows" que proporciona con sus métodos innerWidth() e innerHeight() información sobre el del ancho y de alto respectivamente en píxeles del navegador, esta información permite un desarrollo donde los elementos del juego se muestren proporcionalmente logrando así una adaptabilidad en la visualización del juego al espacio de dibujado.

Estos métodos anteriores se utiliza en:

- En la matriz de proyección durante el cálculo de la perspectiva, logrando que la información renderizada se adapte a la pantalla, evitando así imágenes renderizadas estiradas o estrechas.



Figura 3.23: Ejemplo del mal uso de un campo de visión estático, al reducir o ampliar la pantalla

Como se aprecia en la figura 3.23, usando una proporción de 19/6, en la imagen de la izquierda se muestra de forma correcta, si se reduce la pantalla, se obtiene como resultado la segunda imagen, donde se ve aplanaada, y si aumenta el tamaño de pantalla, la imagen se ve estirada. Para evitar esto se usa la proporción entre el ancho y el alto de la pantalla, `windows.innerWidth / windows.innerHeight`, en el cálculo de la perspectiva.

```
1 perspective(45 * (Math.PI / 180), windows.innerWidth /
  windows.innerHeight , 0.1, 1000);
```

Hay que tener en cuenta que esto lo que hace es adaptar el campo de visión o FOV, por lo que en pantallas pequeñas se verá menos escenario, pero evitaremos el efecto de aplanamiento y lo mismo con pantallas grandes, se verá más pero sin efecto de alargamiento.

- En los elemento a dibujar en el canvas 2d, por ejemplo, si se desea mostrar un elemento en la mitad de la pantalla con hacer el cálculo:

```
1 let posX = windows.innerWidth * 0,5;
 2 let posY = windows.innerHeight * 0,5;
```

Siendo `posX` y `PosY` la posición del elemento en sus coordenadas (`x` , `y`) respectivamente, logramos posicionar el objeto en el centro independientemente del ancho y el largo de la pantalla. Esto permite colocar fácilmente los elementos en la interfaz sin que se vean afectados por el tamaño de la pantalla.

## Interfaz del juego

Otra característica que se ha de tener en cuenta es la cantidad de información que se mostrará en la interfaz del usuario o UI(User Interfaces), es común encontrarse juegos donde la interfaz del usuario está sobrecargada de información que dificulta la correcta visualización de la escena e incluso de la propia interfaz dado que para poder mostrar todo la información se ha de disminuir el tamaño de los elementos, dificultando su lectura. Es por ello que se ha de tener en cuenta varios diseños de interfaces y uso de tamaño de letra según el tipo de dispositivo sobre el cual se esté ejecutando el juego.

En este proyecto se ha realizado dos tipos de diseño de interfaces, una para móviles inteligentes y tablets, que son los que menor tamaño de pantalla presentan y otra que será para el resto, como ordenadores de sobremesa o portátiles.

Primero indicar que elementos se muestran en pantalla:

- Barrada vida, la cual se llena o vacía según los puntos de vida totales y actuales del personaje, además de mostrarse también de forma numérica.
- Un registro o log de los eventos que van sucediendo en el juego, por ejemplo, si se ha cogido un objeto.
- Un minimapa donde se muestra la información de las casillas alrededor del protagonista.
- Una imagen de una poción y una flecha donde se muestra el número de pociones y fechas que tiene el jugador.



Figura 3.24: Interfaz destinada a dispositivos de sobremesa y portátiles

La imagen de arriba corresponde con la interfaz que se muestra en los dispositivos como ordenadores de sobremesa, donde la información queda repartida sin obstaculizar la visualización de la escena.

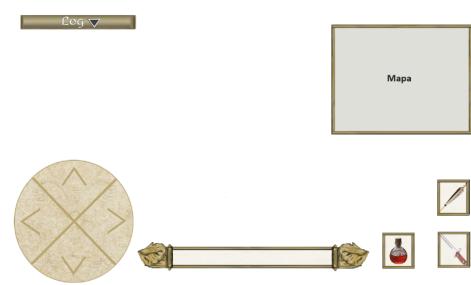


Figura 3.25: Interfaz destinada a dispositivos móviles

La imagen de la figura 3.25 corresponde con la interfaz de dispositivos móviles inteligentes y tablets, dado que en estos dispositivos la entrada de datos es por pantalla se debe incluir más información que en la anterior, como por ejemplo, los botones de dirección y el botón de atacar; Como se aprecia el recuadro del log ha pasado a ser un botón en la parte superior que al pinchar en él despliega el log, haciendo así

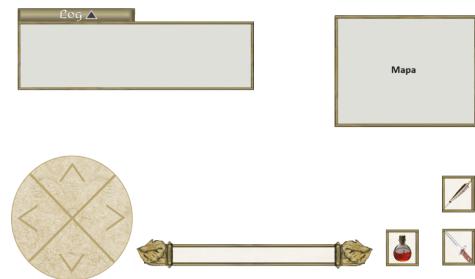


Figura 3.26: Interfaz móvil con la ventana de log desplegado

que sólo se muestre cuando el usuario quiera evitando así la saturación de la interfaz.

La imagen de arriba muestra cómo queda el log desplegado.

Para el tamaño de letra se ha decidido que está se adapte a ancho de la pantalla, en concreto en dispositivos móviles se usa un tamaño de  $(\text{windows.innerWidth} * 0,02)$  y para el resto de dispositivos  $(\text{windows.innerWidth} * 0,01)$ , esta diferenciación es para facilitar la lectura de las letras en dispositivos móviles, dado que queda muy pequeña si se usa la misma para todos los dispositivos.

Para saber sobre tipo de dispositivo se está ejecutando la aplicación se hace uso del objeto JavaScript "Navigator", él cual tiene un atributo denominado userAgent que contiene información sobre el dispositivo anfitrión, entre ellas el nombre del tipo de dispositivo (android, iphone, ipad, etc..).

### Estudio de rendimiento

Este apartado se centra en el estudio de rendimiento del juego desarrollado en distintos dispositivos y navegadores web, para el cálculo de rendimiento se emplea el conteo de fotogramas por segundo (FPS) que es capaz de generar el navegador, además en la comparación se tiene en cuenta los distintos sistemas de luz empleados en el Fragment Shader, en concreto: sin emplear sistemas de luces, empleando sistema de luz ambiental, aplicando sistema de luz ambiental más las luces puntuales.

A continuación se expone una tabla comparativa por cada dispositivo testeado:

- Ordenador de sobremesa, Características:
  - Procesador AMD Ryzen 9 5900X.
  - Tarjeta gráfica RTX 3070ti.
  - Memoria RAM de 16Gb.

- SO Windows 11.

PC sobremesa	Sin luces	Luz ambiental	Luces puntuales
FireFox	33	33	33
Chrome	84	60	60
Edge	80	64	64

Tabla 3.2: Tabla con los resultados en FPS en un dispositivo de sobremesa

Se aprecia en la tabla unos resultados muy buenos, a destacar el navegador FireFox donde se aprecia que no supera de los 30 FPS, lo cual es posible a una limitación del propio navegador.

- Ordenador portátil, Características:
  - Procesador AMD Ryzen 7 5800HS.
  - Gráfica integrada.
  - Memoria RAM de 16Gb.
  - SO Windows 11.

Portátil	Sin luces	Luz ambiental	Luces puntuales
FireFox	19	18	18
Chrome	30	23	22
Edge	29	21	21

Tabla 3.3: portatil

Los resultados obtenidos no son muy buenos, en la mayoría de los casos con fotogramas por segundo por debajo de los 30, se entiende que es debido a que es un portátil destinado a ofimática y no para juegos. Comparando el resultado entre los distintos navegadores web se aprecia una diferencia considerable entre el navegador FireFox del resto.

- SmartPhone Samsung A35:
  - Procesador Exynos 1380
  - Gráfica integrada.
  - Memoria RAM de 8Gb.
  - Android 14.

Tabla 3.4: Tabla con los resultados en FPS en un dispositivo móvil Samsung

Samsung A35	Sin luces	Luz ambiental	Luces puntuales
FireFox	25	26	24
Chrome	33	20	20

SmartPhone de gama media, se aprecia en la tabla que no da buenos en los cálculos de luz, destacando positivamente en el navegador Chrome, donde sin luces da tasas superiores a los 30FPS.

- Tablet lenovo M11:
  - Procesador MediaTek Helio G88
  - Gráfica integrada.
  - Memoria RAM de 8Gb.
  - Android 14.

Table Lenovo	Sin luces	Luz ambiental	Luces puntuales
FireFox	7	5	NR
Chrome	6	5	NR

Tabla 3.5: Tabla con los resultados en FPS en un portátil marca Lenovo

Dispositivo tablet de gama baja, da resultados muy negativos haciendo imposible jugar en él.

- Odin 2:
  - Procesador Snapdragon 8 Gen 2
  - Gráfica Adreno 740 GPU
  - Memoria RAM de 12Gb.
  - Android 14.

ODIN 2	Sin luces	Luz ambiental	Luces puntuales
FireFox	40	37	30
Chrome	60	47	47

Tabla 3.6: Tabla con los resultados en FPS en un dispositivo móvil Odin 2

Dispositivo Android destinado a juegos, se aprecian unos resultados muy buenos, destacando positivamente el navegador Chrome.

- Iphone 13:
  - Apple A15 Bionic
  - Gráfica integrada

Iphone13	Sin luces	Luz ambiental	Luces puntuales
FireFox	133	80	NR
Safari	137	80	NR

Tabla 3.7: Tabla con los resultados en FPS en un iphone

- Memoria RAM de 4Gb.
- IOS 17.5.



## Capítulo 4

# Resultados y discusión

La finalización de este TFM a permitido cumplir con todos los objetivos marcados al principio de este documento [1.2 \(Objetivos\)](#), donde:

- Se ha desarrollado una aplicación multiplataforma web mediante el uso de HTML5 y TypeScript.
- Se ha adquirido los conocimientos necesarios para poder aplicar renderizados 3D a la aplicación web gracias al uso de WebGL.
- Se ha logrado que la aplicación sea portable a distintos tipos de dispositivos, teniendo en consideración características como entrada de datos y tamaño de pantalla.
- Se ha conseguido combinar todo lo anterior con éxito para el desarrollo de un juego multiplataforma orientado a web.

A continuación se muestra el juego resultante mediante capturas de pantallas del mismo:



Figura 4.1: Juego resultante ejecutandose en un navegador de PC



Figura 4.2: Juego resultante ejecutandose en un navegador de Ipad

## Capítulo 5

# Conclusiones

Las conclusiones que se han llegado durante la realización de este proyecto son las siguientes:

Es factible la creación de aplicaciones donde se use el navegador web para mostrar imágenes renderizadas 3D, esto es principalmente gracias a la gran estandarización existente entre los distintos navegadores web, aún así, hay que tener en cuenta las peculiaridades de cada uno, como por ejemplo, algunos navegadores como Firefox tienen por defecto limitada la tasa de refresco a 30 fotogramas por segundo, otros como Safari para Iphone no permite en sus políticas de seguridad dejar que las aplicaciones web puedan ponerse en pantalla completa (exceptuando algunos casos como por ejemplo videos), pero el mismo navegador para Ipad si que deja, diferencias existentes entre eventos de entrada, por ejemplo, en Android no se diferencia entre el evento de pulsación de ratón y el de pulsación de la pantalla, mientras que en dispositivos de Apple si, diferencias de rendimiento donde algunos navegadores gestionan mejor la ejecución de la aplicación, etc.. Por lo tanto, hay que ir mirando casos concretos en cada navegador para lograr una aplicación realmente portable y multiplataforma.

También comentar la gran cantidad de documentación que existe para el desarrollo aplicaciones web, haciendo que la búsqueda y contraste de la información sea sencilla, pero por otro lado, hay poca documentación sobre el desarrollo para WebGL, y menos aún para su versión 2, por lo que ha resultado un poco difícil poder acudir a otras fuentes si en los principales no quedara algún concepto claro.

Por último indicar que al ejecutarse el código en el navegador del cliente todo el código del proyecto queda expuesto, por lo que habrá que recurrir a métodos como la ofuscación para dificultar que el código pueda ser copiado sin permiso del autor.



# Bibliografía

- [1] T. K. Group, “Opengl 4.6 at a glance.” <https://www.khronos.org/opengl/>. [Accessed 19/01/2024].
- [2] Microsoft, “getting started with direct3d.” <https://learn.microsoft.com/es-es/windows/win32/getting-started-with-direct3d>. [Accessed 19/01/2024].
- [3] GPUOpen, “Vulcan.” <https://gpuopen.com/vulkan/>. [Accessed 19/01/2024].
- [4] I. Apple, “Metal overview.” <https://developer.apple.com/metal/>. [Accessed 05/09/2024].
- [5] Three.js, “fundamentals of threejs.” <https://threejs.org/manual/#en/fundamentals>. [Accessed 19/01/2024].
- [6] SDL, “About sd.” <https://www.libsdl.org/>. [Accessed 19/01/2024].
- [7] LibGDX, “Libgdx.” <https://libgdx.com/>. [Accessed 19/01/2024].
- [8] LWJGL, “Lwjgl.” <https://www.lwjgl.org/>. [Accessed 19/01/2024].
- [9] P. S. Inc, “phaser3.” <https://phaser.io/tutorials/getting-started-phaser3>. [Accessed 19/01/2024].
- [10] Pygame, “about pygame.” <https://www.pygame.org/wiki/about>. [Accessed 19/01/2024].
- [11] D. C. Farhad Ghayour, *Real-Time 3D Graphics with WebGL2, Second Edition*. Packt, 2018.
- [12] J. K. b. L.-K. EDave Shreiner, Graham Sellers, *Opengl Programming Guide, Eighth Edition*. Addison-Wesley, 2013.
- [13] J. de Vries, *Lean OpenGL - Graphics Programming*. Kendall Welling, 2020.
- [14] K. Matsuda and R. Lea, *WebGL Programming Guide*. Addison-Wesley, 2013.

- [15] A. Rauschmayer, *JavaScript por impatient programmers*. Independently published, 2019.
- [16] Mozilla, “¿qué es javascript?.” [https://developer.mozilla.org/es/docs/Learn/JavaScript/First\\_steps/What\\_is\\_JavaScript](https://developer.mozilla.org/es/docs/Learn/JavaScript/First_steps/What_is_JavaScript). [Accessed 19/01/2024].
- [17] Typescriptlang, “Typescript for the new programmer.” <https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html>. [Accessed 19/01/2024].

