Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Proyecto de Fin de Máster

Posibilidades de desarrollo de videojuegos FPS utilizando tecnología WebGL

Autor:

Guillem Aguado Sarrió

Tutor Académico:

Roberto Vivó Hernando

Curso Académico: 2015/2016

Resumen

Las tecnologías web han evolucionado mucho y ahora se ha conseguido renderizar gráficos 3D interactivos en el propio navegador, sin necesidad de plug-ins. Además utilizando la tarjeta gráfica del propio sistema de quien accede a ellos por medio de un navegador.

Esto ha suscitado mucho interés por parte de la comunidad mundial de programadores, de modo que han surgido numerosas librerías para explotar las posibilidades de la nueva tecnología WebGL.
Entre estas librerías, Three.js parece haber logrado cierto grado de popularidad entre la comunidad y en número de desarrolladores y usuarios.

Es por todo esto que surge la motivación de este proyecto de crear un videojuego de exploración e interacción de tipo FPS utilizando WebGL y Three.js para poder ver el alcance y las posibilidades que esto ofrece hoy en día, y además evaluar prestaciones sobre diferentes navegadores y sistemas.

Además, el hecho de desarrollar un videojuego y explorar nuevas tecnologías gráficas en 3D dota de mucha experiencia al propio desarrollador, que en un futuro servirá para o bien desarrollar nuevos proyectos, o bien desenvolverse en el mercado laboral con éxito.

En el presente proyecto se exponen las tecnologías utilizadas, tanto en el diseño y modelado de contenido como en la propia implementación y publicación en la red de un juego FPS, así como las pruebas realizadas sobre distintos navegadores en ciertos sistemas operativos.

Palabras clave:

WebGL: Especificación para el desarrollo y ejecución de gráficos 3D en navegador.

Three.js: Librería gráfica para la generación de gráficos 3D en navegador.

FPS: Género de videojuegos basado en combate mediante armas o proyectiles desde una perspectiva en primera persona.

Rendimiento: Producto o utilidad que da una persona o entidad.

Javascript: Lenguaje de script que permite desarrollar la lógica de una aplicación web.

Blender: Programa de modelado y animaciones en 3D libre.

3D Studio Max: Programa de modelado y animaciones 3D de pago.

GIMP: Programa de manipulación y generación de imagen libre.

Audacity: Programa para manipulación de piezas de audio libre.

Keywords:

WebGL: An specification for creating and running 3D graphics on the browser.

Three.js: 3D library for generating 3D graphics on the browser.

FPS: Videogame gender based on combat using guns or projectils in a first person perspective.

Performance: Product or utility given from a person or an entity.

Javascript: Scripting languaje that allows to create the logic of a web application.

Blender: A free software for modelling and animating models on 3D.

3D Studio Max: A propietary software for modelling and animating models on 3D.

GIMP: Free software for creating and manipulating images.

Audacity: Free software for manipulating audio files.

Índice de la memoria

Parte 1

Motivación y objetivos

1.1. Motivación y contexto tecnológico del proyecto.

En los últimos tiempos, la tecnología web, que siempre ha sido un importante sector en la informática, ha experimentado un crecimiento importante. Con la introducción de WebGL, se abren las puertas al desarrollo de contenidos 3D interactivos, del mismo modo que se pueden desarrollar en una aplicación de escritorio, pero únicamente usando tecnología web.

Teniendo esto en cuenta, y dado el gran abanico de posibilidades que este desarrollo despierta, el presente proyecto tiene cabida al explorarlas y explotarlas para realizar una aplicación interactiva en 3D que muestre algunas características que se pueden desarrollar y otras que no, y si es o no complejo desarrollarlas con la tecnología actual.

Se puede afirmar con seguridad que el mercado de las aplicaciones interactivas y de los videojuegos es muy lucrativo e interesante hoy en día, y en aras de tratar de encontrar empleo es muy útil tener experiencia en el campo. Teniendo en cuenta esto y dado que la web es un mercado muy explotado, este proyecto tiene también un efecto muy positivo en materia de proporcionar experiencia de desarrollo de aplicaciones interactivas en el web, y con las tecnologías nuevas y populares como son la librería gráfica Three.js y el propio WebGL.

Se pretende aprender mas sobre la librería Three.js y el funcionamiento de WebGL, al mismo tiempo que se descubren las flaquezas y virtudes de una tecnología que es nueva y que aun esta en desarrollo, de modo que no está perfeccionada. Esto es un doble beneficio que interesa tanto al desarrollador del proyecto como a un posible futuro desarrollador de aplicaciones, o que desee crear un proyecto en esta misma línea, algo bastante posible dado que la naturaleza de esta tecnología es siempre seguir creciendo dadas sus características y su ámbito comercial que tiene tendencia a ser muy explotado en el mercado.

El hecho de crear una aplicación 3D interactiva es muy beneficioso también, dado que son de las mas consumidas en el mercado y esto de cara al futuro y a encontrar trabajo es bastante útil, con objeto de convencer a un empleador o de vender un producto similar en plataformas como páginas web con juegos o *steam*, por ejemplo.

1.2. Objetivos del proyecto.

El proyecto tiene dos objetivos principales, aprender a desarrollar aplicaciones interactivas 3D con tecnologías web novedosas como son WebGL y Three.js, y descubrir las posibilidades y limitaciones que ofrecen hoy en día en las principales plataformas, como son el navegador Chrome de Google, Firefox, u otros en distintos sistemas operativos como son Windows o Linux.

Mas concretamente, se pretendía cubrir ciertos objetivos mas particulares, los cuales son:

- **Diseño de los elementos de la aplicación.**
  - Diseño de los escenarios y sus características.
  - Diseño de los personajes.
  - Diseño de la interfaz de juego(teclado o ratón).
  - Diseño de la interfaz de ayudas visuales de la aplicación(textos, barras de salud y otras características).
  - Diseño de la banda sonora y efectos.
- **Modelado y generación de contenido multimedia.**
  - Búsqueda de contenidos gratuitos por la web.
  - Modelado en Blender y 3D Studio Max de animaciones y características aplicados a modelos realizados por artistas.
  - Ajuste de piezas de audio y efectos mediante programas de edición de audio.
- **Implementación.**
  - Creación del código correspondiente a la lógica de la aplicación, y al de su interfaz gráfica/escenarios.
  - Carga de modelos estáticos y animados, carga de piezas de música y sonidos.
  - Corrección de *bugs* o fallos y mejoras.
- **Testing.**
  - Pruebas de rendimiento bajo distintos navegadores y sistemas.
  - Pruebas de fluidez y usabilidad.

1.3. Estructura de la memoria.

En este punto de la memoria se han desarrollado las motivaciones y el contexto tecnológico que mueven el presente proyecto. También se han citado los objetivos principales y particulares del mismo.

En la segunda parte, se expondrán las tecnologías utilizadas, el motivo de haberlas utilizado y sus características, además también se hará referencia al estado del arte en el desarrollo de aplicaciones interactivas 3D como es el caso de la desarrollada en el presente proyecto.

En la tercera parte, se explicará detalladamente todo el proceso de diseño, modelado e implementación de todas las partes de la aplicación, que incluye los escenarios, los personajes y sus modelos, animación u inteligencia artificial en su caso así como la lógica y módulos software que mueven a la propia aplicación.

En la cuarta parte se pasará a explicar los experimentos realizados en diferentes navegadores y sistemas, como son Chrome, Firefox(navegadores), y Windows o Linux(sistemas).
Se pretende medir el rendimiento, tiempo de carga y fluidez o usabilidad de los diferentes navegadores en diferentes sistemas operativos para poder realizar una comparativa.

En la quinta parte, se expondrán las conclusiones extraídas del desarrollo, implementación y pruebas realizadas a lo largo de todo el proyecto.

También se hará un apunte acerca de posible trabajo futuro en esta misma línea, o incluso aprovechando el trabajo realizado durante el proyecto.

Parte 2

Tecnologías y estado del arte

2.1. Tecnologías utilizadas.

2.1.1. WebGL.

WebGL es una especificación estándar que está actualmente en desarrollo y que se encuentra gestionado en la actualidad por el consorcio tecnológico Khronos Group.

Su utilidad es la de poder mostrar gráficos 3D en el navegador acelerados por hardware mediante el uso de la unidad de procesamiento gráfico(GPU), sin necesidad de plug-ins. Esto seria posible en cualquier plataforma que soporte OpenGL 2.0 u OpenGL ES 2.0.

WebGL empezó en los experimentos realizados con Canvas 3D en el seno de Mozilla.

En 2006 se mostró un prototipo de Canvas 3D y a finales del año siguiente Mozilla y Opera habían hecho sus propias implementaciones. Finalmente a inicios del año 2009 Mozilla y el consorcio Khronos formaron el WebGL Working Group.

El hecho de que permita crear gráficos en 3D y aceleración por GPU, además de ser una especificación en desarrollo hacen que sea interesante para ser seleccionada para un proyecto como el presente, para así poder estudiar sus posibilidades.

WebGL utiliza el elemento Canvas HTML5 y accede mediante interfaces Document Object Model(DOM). Carece de las rutinas matemáticas de matriz eliminadas en OpenGL 3.0.

Este estándar funciona bastante bien a la hora de desarrollar aplicaciones 3D interactivas, dado que la aceleración por hardware y otras características como poder crear efectos de iluminación fácilmente le dan un buen rendimiento respecto a otras tecnologías, y aunque está aun bastante limitado dado que solo cuenta con los shaders de vértice y fragmento(pequeños programas que se ejecutan en la GPU), y no computación y geometría o tesselación como openGL 4, da bastante mas versatilidad para el desarrollo de este tipo de aplicaciones.

Hoy en día funciona bajo Google Chrome, Internet Explorer(versión 11), y Mozilla Firefox, aunque también en computadores de sobremesa bajo el sistema operativo de Apple y con limitaciones en los navegadores Opera browser y Safari.

En cuanto al aspecto de la seguridad informática, es cierto que se han detectado fallas en la seguridad a la hora de usar WebGL en el navegador.

El hecho de poder ejecutar el código del o de los componentes WebGL en la unidad de procesamiento gráfico permite que este código pueda aprovecharse de vulnerabilidades del sistema y congelarlo o hacer que caiga.

No obstante, un portavoz de Google ha declarado que muchos procesos de WebGL, incluido el de la GPU, se ejecutan por separado y dentro del sandbox de Chrome(un sandbox es un entorno de pruebas donde se ejecuta código para probarlo).

El grupo Krhonos ha afirmado ademas que están evaluando las advertencias sobre seguridad y que los fabricantes de GPUs están incorporando soporte para un mecanismo que debería ayudar a solventar el problema de seguridad.

2.1.2. Three.js.

Debido a que WebGL es una especificación que trabaja a bajo nivel, ya que trabaja directamente sobre la GPU, su codificación es mas compleja que otros estándares web.

No obstante, para poder crear aplicaciones sin trabajar a tan bajo nivel han surgido numerosas bibliotecas de JavaScript que hacen la tarea mucho mas sencilla. Aquí se citan conocidas librerías:

C3DL, CopperLicht, Curve3D, CubicVR, EnergizeGL, GammaJS, GLGE, GTW, JS3D, Kuda, O3D, OSG.JS, PhiloGL, Pre3d, SceneJS, SpiderGL, TDL, Three.js, X3DOM. BabylonJS.

Entre todas ellas, Three.js es la mas popular en numero de usuarios, aunque también es ligera y la complejidad es mucho menor que la especificación WebGL, y es por estos motivos por los que se ha seleccionado como librería para la realización del presente proyecto.

Si bien es cierto que BabylonJS ha ganado también popularidad debido a que ha sido desarrollada por miembros de Microsoft.

Es una biblioteca que fue creada y liberada en la plataforma GitHub por el español Ricardo Cabello en abril de 2010. Inicialmente creada en ActionScript y finalmente traducida a JavaScript dado que de este modo no era necesario compilar antes de cada carga y además proporciona independencia de plataforma.

Puede utilizarse conjuntamente para mostrar gráficos 3D con el elemento Canvas de HTML5, SVG(Scalable Vector Graphics), o bien con WebGL.

Se ha popularizado como librería para la generación de aplicaciones con gráficos 3D animadas e incluso interactivas.

Entre las características de Three.js podemos encontrar por ejemplo:

- Renderizadores: Canvas, SVG y WebGL.
- Efectos: Anaglifo, bizco y barrera de paralaje.
- Animación: armaduras, cinemática directa, cinemática inversa, morphing y fotogramas clave.

- Luces: ambiente, dirección, luz de puntos y espacios, sombras: emite y recibe.
- Materiales: Lambert, Phong, sombreado suave, texturas y otras.
- Cargadores de datos: binario, imagen, JSON y escena.
- Depuración: Stats.js, WebGL Inspector, Three.js Inspector.

Podemos encontrar mas de 150 archivos de códigos de ejemplo mas las fuentes, modelos, texturas, sonidos y otros archivos soportados en threejs.org, lo cual resulta muy útil a la hora de crear código, sobretodo teniendo en cuenta las variaciones sobre la librería que se efectúan día a día.

La biblioteca puede ser utilizada mediante un enlace al archivo JavaScript local o remoto en el archivo html de la página:

<script src="js/three.js"></script>

A partir de este momento tenemos acceso en nuestro código JavaScript a todas las funciones y utilidades de la librería. Como ejemplo podemos tener lo siguiente:

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

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

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

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

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

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

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

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

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

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

   render();

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

Figura 1.- Fragmento de código presente en el Wikipedia en el documento sobre three.js.

Que lo que hace es crear una escena y añadir una cámara, un cubo y una luz a la escena. Crea además un renderizador WebGL y añade su viewport al elemento body del documento. Cuando ha cargado, el cubo rota en sus ejes X y Y. Puede servirnos como ejemplo para empezar a crear nuestras primeras escenas con la librería, pero habría que profundizar mucho mas en la documentación para crear escenas mucho mas complejas.

2.1.3. Tween.js.

La librería JavaScript Tween.js proporciona una poderosa y sencilla interfaz de *tweening* de propiedades numéricas de objetos y también propiedades de estilo CSS. También permite encadenar *tweens* para crear secuencias complejas.

El *Tweening*, que es la forma acortada del término *in-betweening*, o intermediación, es el proceso de crear imágenes o propiedades intermedias entre dos situaciones o imágenes clave dadas para dar la sensación de que la primera se convierte suavemente en la segunda.

Este motor se ha utilizado para generar el efecto de que las partículas utilizadas durante el desarrollo del proyecto se mueven de un sitio a otro de una manera u otra o cambian su escala de mayor a menor, por ejemplo, entre dos situaciones clave dadas. Esto ha sido la clave del desarrollo de los efectos en las partículas que dan imagen a los disparos en el juego.

2.1.4. Servidor de la Universidad Politécnica y servidor de test.

Se han utilizado dos servidores para la creación de la aplicación, uno que contenía la versión de pruebas en constante actualización, en local, y otro que contenía la última versión estable publicada.

Para el servidor de pruebas se ha utilizado un servidor WampServer, que es un entorno de desarrollo web bajo Windows.

El acrónimo WAMP utilizado en el nombre del entorno WampServer describe a una infraestructura de red que utiliza las siguientes tecnologías:

- Windows, como el sistema operativo en el que se ejecuta.
- Apache, el servidor.
- MySQL, el gestor de bases de datos.
- PHP, el lenguaje de programación.

El uso de un sistema WAMP permite subir páginas web a Internet y gestionar datos en ellas. En nuestro caso, toda la gestión y los datos se encuentran en local, de modo que el servidor WampServer crea una carpeta www en el sistema de archivos, la cual contiene todos los datos de nuestra aplicación web en local y podemos acceder a ella mediante el enlace:

http://localhost/proyecto.html, siendo proyecto.html el nombre hipotético que le hubiésemos dado a la página principal de la aplicación, por ejemplo.

Para el caso del servidor donde finalmente se aloja la aplicación en su última versión estable y que es público en Internet, se ha utilizado el servidor de la Universidad Politécnica de Valencia, donde se han ubicado los archivos correspondientes a la última versión sobre la que se han realizado test y comprobado que es estable.

Dicho servidor se encuentra disponible desde el enlace:

http://personales.alumno.upv.es/guiagsar/tfmsn/proyectoGPC.html

Dicho servidor sirve como copia de seguridad de los archivos de la aplicación, ya que se tiene la certeza de que es una versión estable de la misma.

Cualquier usuario de la red con acceso a ella puede acceder a la aplicación en cualquier momento usando este último enlace.

2.2. Estado del arte.

El estado del desarrollo de videojuegos en HTML5 ha evolucionado mucho en poco tiempo, los desarrolladores han pasado de ser muy cautelosos con la tecnología a entrar en un frenesí incontrolable, de modo que cada día salen mas y mas juegos decentes en 2D y 3D.

Recientemente, se ha conseguido portar un juego de realidad virtual(VR), llamado SECVRITY, inicialmente creado en el motor de videojuegos Unity a WebVR, una API experimental en JavaScript que permite acceso a dispositivos de realidad virtual en tu navegador, tales como son el Oculus Rift o Google Cardboard.

WebVR esta aun en etapas tempranas de su existencia, y se encuentra aun en desarrollo, además de que solo está disponible en la versión última compilada de Firefox, compilaciones experimentales de Chrome o en el navegador de Internet de Gear VR, llamado *Samsung Internet for Gear VR*.

No obstante, y a pesar de que aun no hay formas nativas de portar juegos hechos en motores como Unity a WebVR, se ha encontrado un modo de hacerlo.

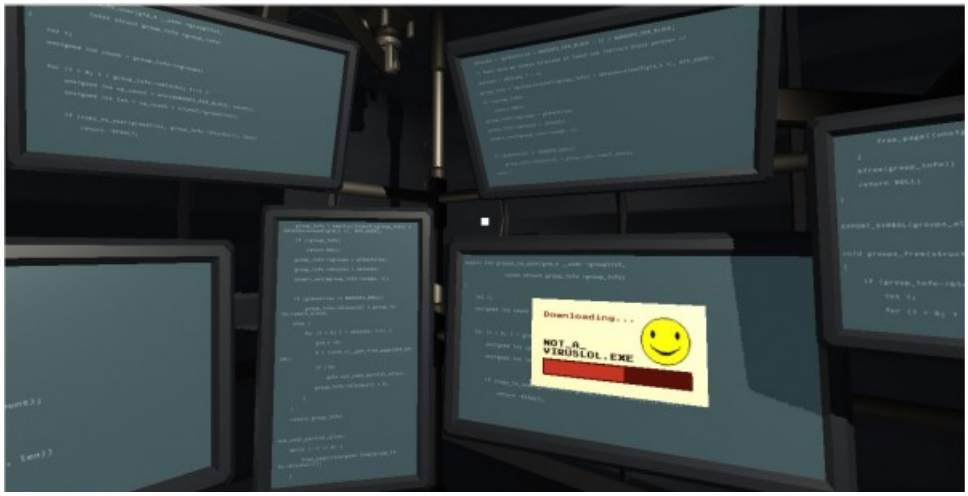


Figura 2.- SECVRITY, un juego creado en Unity y portado a WebGL.

Se han podido realizar también simulaciones utilizando el Vertex Shader con 100.000 bits de información sin que el navegador se colapse. La prueba de ello es la que se encuentra en el siguiente enlace:

http://visualiser.fr/Babylon/cloud/index.html,

Sin embargo, y a pesar de todo esto y de que la comunidad se haya volcado en la creación de librerías JavaScript para WebGL y en la creación de juegos, parece que aun hay una reticencia a creer que los juegos en HTML5, o en definitiva, en la web, tengan tan buen rendimiento como un juego de escritorio o de consola.

El hecho de que muchos juegos se porten de otras plataformas de desarrollo como Unity, es significativo de que es duro aun crear juegos desde la nada y es muy sencillo portarlos desde un lugar donde se sabe que es mas sencillo crearlos y funcionan bien, sin embargo esto hace que no se vean igual probablemente o sufran en parte esta portación desde otra plataforma.

A pesar de todo parece que la tecnología evoluciona rápido y surgen nuevas versiones de las librerías JavaScript día a día, así como juegos cada vez mejores, por lo que no es descabellado pensar que en un futuro próximo los juegos HTML5 en navegador utilizando WebGL o incluso WebVR para dispositivos de realidad virtual puedan ser tan competitivos como otros juegos de plataformas consolidadas durante años como los juegos de escritorio para computador o de consola.

También se puede usar los juegos en HTML5 para el marketing, y eso han hecho los miembros de 20th Century Fox, creando un juego en HTML5 que promociona su nueva película, Ice Age: Collision Course.

Figura 3.- Ice Age Collision Course Planet Pool, videojuego de navegador en HTML5.

Parte 3

Diseño, modelado e implementación de la aplicación y sus componentes

3.1. Diseño de la aplicación.

3.1.1. Arquitectura de la aplicación.

La aplicación esta formada por varios módulos independientes, que trabajan de forma coordinada y utilizan diversas tecnologías para funcionar:

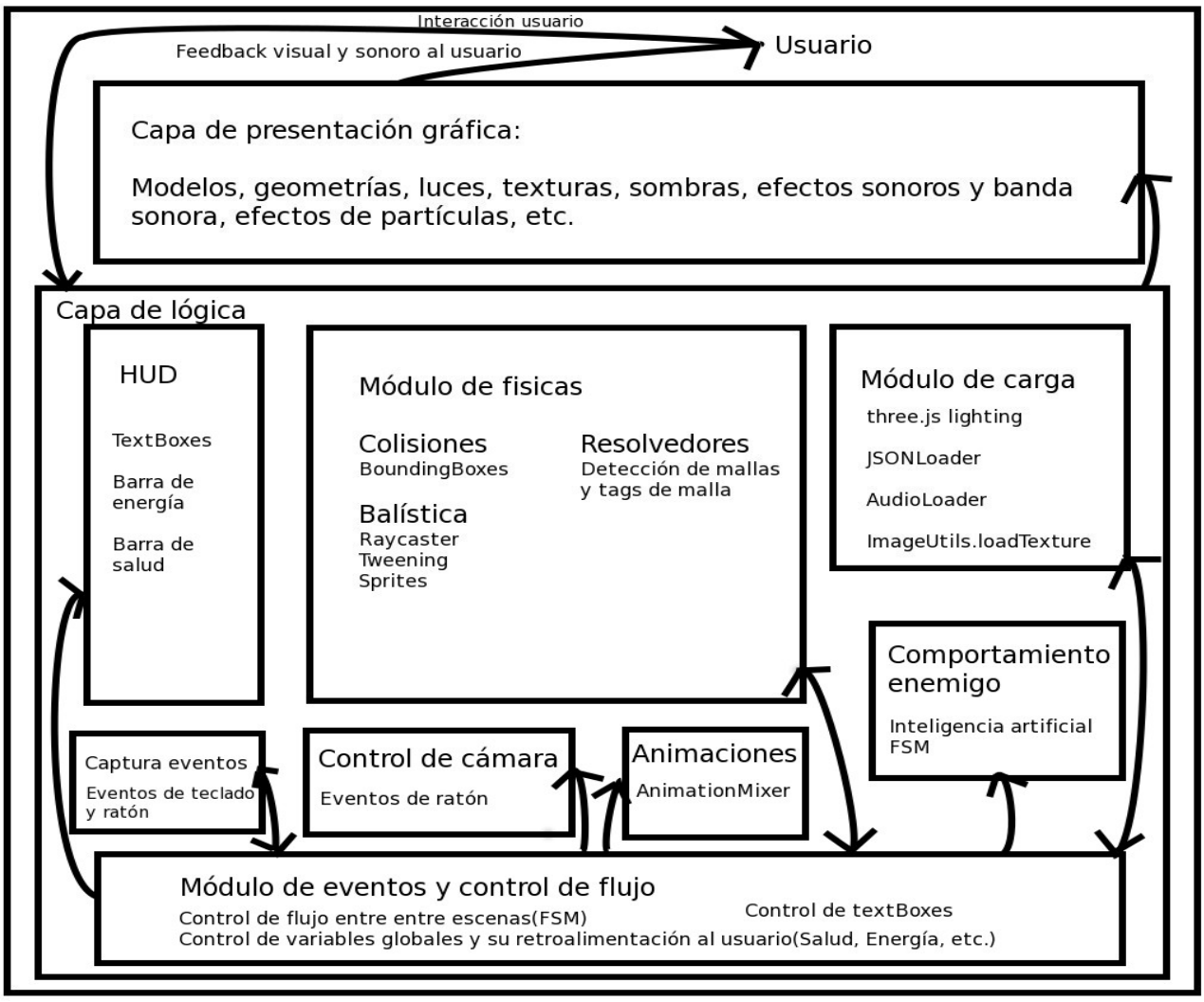


Figura 4.- Arquitectura de la aplicación en capas y sus módulos.

3.1.2. Diseño de los módulos.

Los diferentes módulos de la aplicación han sido diseñados por separado, los diseños realizados son los siguientes:

3.1.2.1. Capa de presentación.

La capa de presentación se basa en mostrar gráficos 3D mediante el elemento Canvas al usuario, además de feedback sonoro y música.

Se compone de dos escenas diferentes que están construidas mediante un terreno, una caja de fondo o *Skybox*, y diferentes elementos gráficos que están dentro de la propia escena, además de los efectos sonoros y música.

Estos elementos gráficos son de diferentes tipos, entre los cuales podemos encontrar sombras arrojadas, modelos 3D, figuras 3D y modelos animados.

Todos estos elementos aparecen o desaparecen y son controlados por el control central de la lógica de la aplicación en el motor de eventos y control de flujo.

Figura 5.- Captura de la aplicación en la que se ve la primera escena o las “Llanuras”, se aprecian varios elementos con sombras arrojadas y el personaje principal, además del elemento del HUD “Barra de energía”.

Esta campa proporciona la realimentación principal al usuario, tanto acústica como visual, y lo guía por el flujo de la historia y por las diferentes situaciones que debe afrontar. También se incluyen pantallas de carga entre escenas en las que se muestra una caja de texto que lo indica.

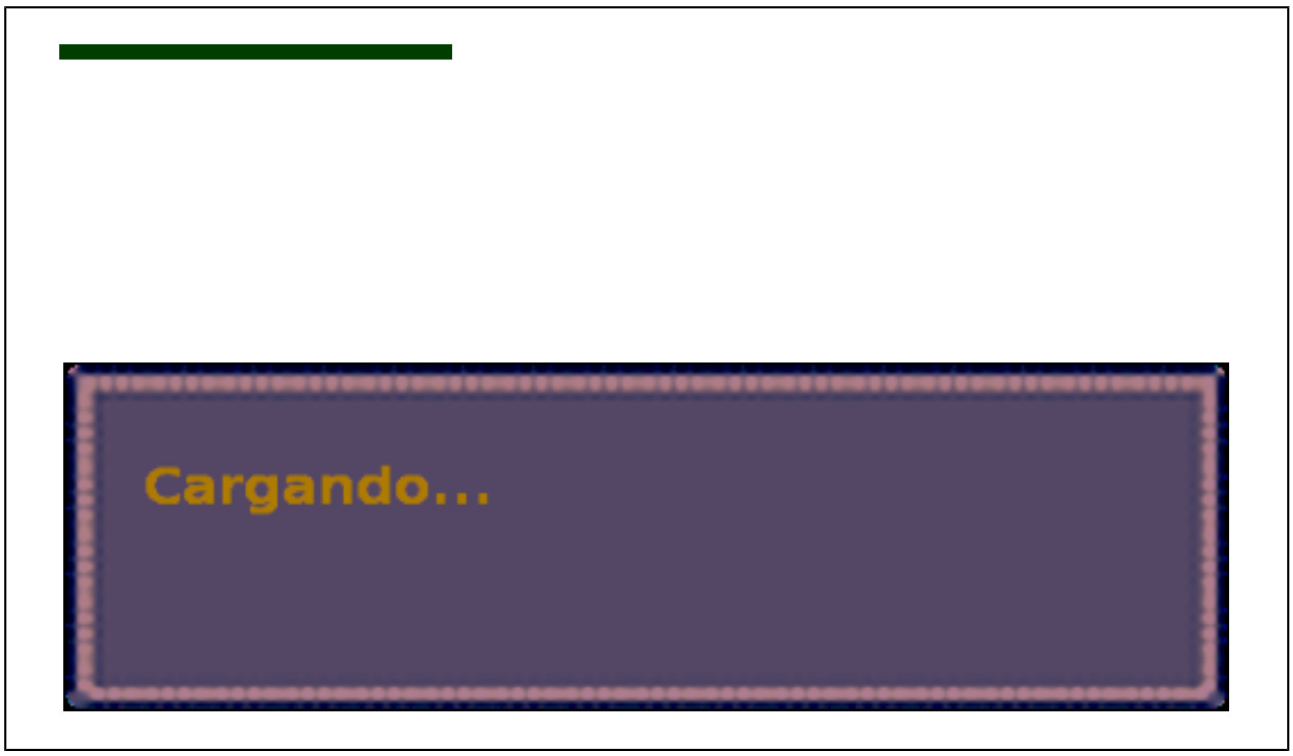


Figura 6.- Captura de la aplicación en la que se aprecia la escena de carga.

Todas las formas de interacción de teclado y ratón estan también explicadas mediante este sistema de cajas de texto, que además guía al usuario en su experiencia interactiva y le ofrece información sobre la propia historia o trama del juego.

3.1.2.2. Capa de lógica.

La capa de lógica es la que realiza todos los cálculos, tiene los algoritmos y usa las tecnologías necesarias para hacer que la aplicación funcione y pueda mostrar los gráficos y audio que el usuario percibe en la capa de presentación, además también recoge la interacción del usuario con el ratón y teclado o *touchpad* para utilizarla en la propia lógica como información que se usará para generar el *frame* correspondiente.

La lógica se compone a su vez de muchos módulos, independientes entre ellos excepto por el módulo que ejerce el control centralizado, que es el de eventos y control de flujo de la aplicación, el cual interactúa con todos ellos y a veces también recibe realimentación de los mismos.

Módulo de físicas: El módulo de físicas se compone de tres elementos:

1. Colisiones: Es la parte de la lógica correspondiente a las funciones que calculan colisiones entre cajas de inclusión de objetos físicos de la escena.
2. Balística: Es la lógica correspondiente a los disparos y al código que permite que se puedan pintar y calcular la posición del proyectil y sus partículas, así como tamaño de las mismas, etc.
3. Resolvedores: Corresponde a las funciones que se ejecutan cuando un proyectil alcanza una malla, de modo que desencadenan algún tipo de comportamiento para resolver este hecho.

Figura 7.- Captura en la que se observa al personaje colisionando con una roca en la escena, al tiempo que lanza un rayo azul al pedestal a su lado y el rayo lo alcanza.

Figura 8.- Efecto expansivo de un rayo colisionado con un objeto(animación de partículas). En esta captura se ha desactivado la magia del usuario tras disparar pulsando la tecla M, para poder apreciar bien el efecto de la propia colisión del rayo.

Módulo de carga: Este módulo sirve para cargar las mallas, modelos, audio y diferentes texturas y elementos en la escena. Es un módulo básico que toda aplicación gráfica debería poseer. Este módulo realiza todas las cargas e introduce las mallas, audio y elementos en la escena.

Módulo de captura de eventos: Es el encargado de recoger la interacción del usuario con la aplicación mediante el teclado, ratón y *touchpad.*

Existen diferentes eventos que se pueden capturar y que la aplicación tiene en cuenta, como son el evento de mover el ratón o deslizar el dedo por el *touchpad* para el control de cámara(el cual es otro módulo que es el de control de la cámara precisamente), click con el botón principal o con el *touchpad* para los disparos y eventos de pulsado de teclas para diferentes funcionalidades como por ejemplo activar o desactivar la magia o avanzar y retroceder.

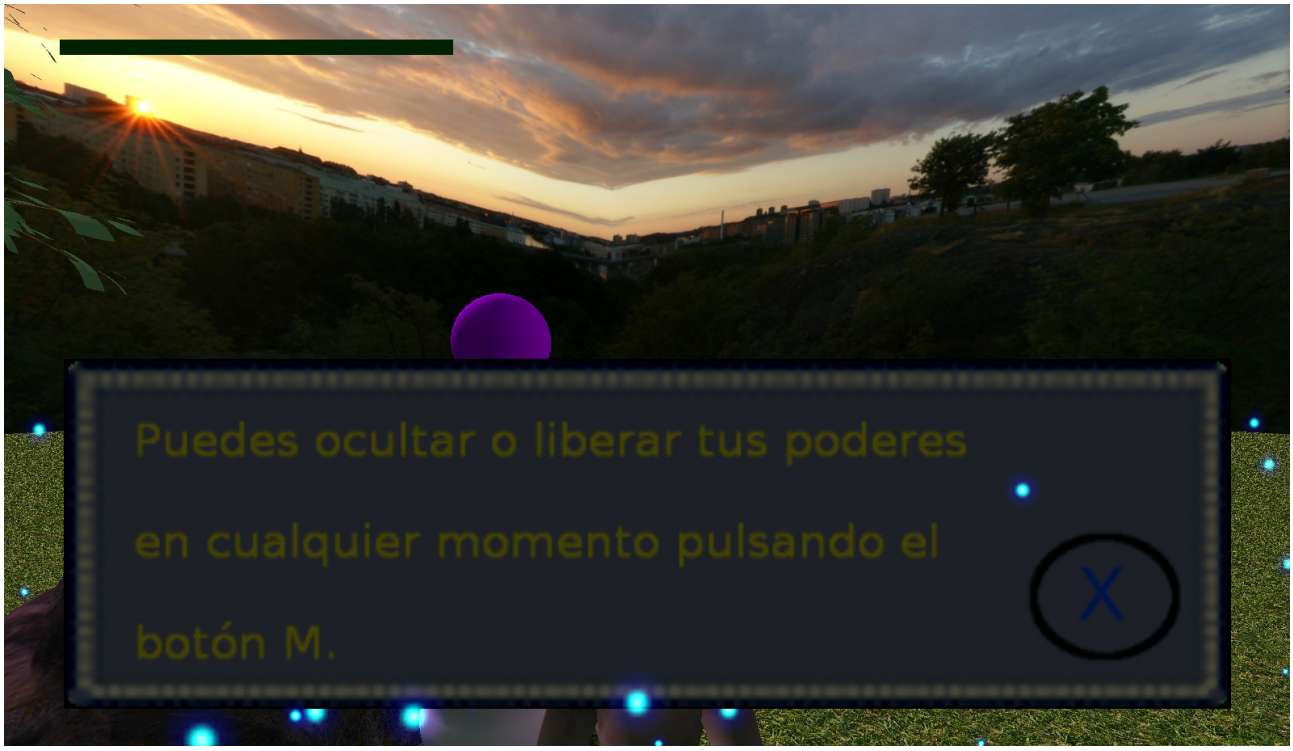


Figura 9.- Explicación de uno de los carteles de la aplicación de uno de los eventos de teclado, mientras se expone a la derecha que se puede activar otro para cerrar el cartel con la tecla X.

Módulo de control de cámara: Este módulo te permite manipular tu dirección al tiempo que mueves la propia cámara y al personaje, funciona o bien por control de ratón o bien por *touchpad.*

Es un módulo auxiliar que funciona junto con el módulo de captura de eventos para completar la interacción del usuario con la aplicación.

Módulo de animaciones: Este módulo es el que contiene las funciones que reproducen animaciones y que son actualizadas en la función de *render* de la aplicación.

Módulo del HUD(Heads-Up Display): Este módulo controla los elementos del HUD, que son elementos gráficos de ayuda visual que se muestran delante de todos los demás elementos gráficos que aparecen en la escena vistos por la cámara.

Posee herramientas para mostrar carteles o no y situarlos en su posición correcta, o para mostrar barras de energía y salud con mas o menos divisiones(las barras del HUD tienen 100 elementos o divisiones individuales cada una).

Módulo de comportamiento enemigo: Este módulo contiene la inteligencia artificial del enemigo principal del juego y su máquina de estados finitos(FSM), que controla su comportamiento.

Una Máquina de Estados Finitos o FSM es un modelo de computación utilizado para diseñar programas y también circuitos lógicos secuenciales. Es un modelo abstracto, y tiene la característica de tener un conjunto de estados posibles delimitado.

La máquina solo puede estar en un estado a la vez, al cual se llama estado actual, pero puede pasar a otros estados mediante el uso de condiciones o *triggers.*

Estas máquinas se utilizan en juegos para definir comportamientos, de modo que una entidad o el propio juego se encuentran en un estado pero pueden transitar a otros estados mediante un *trigger.*

Este es un esquema de la FSM del comportamiento del rival:

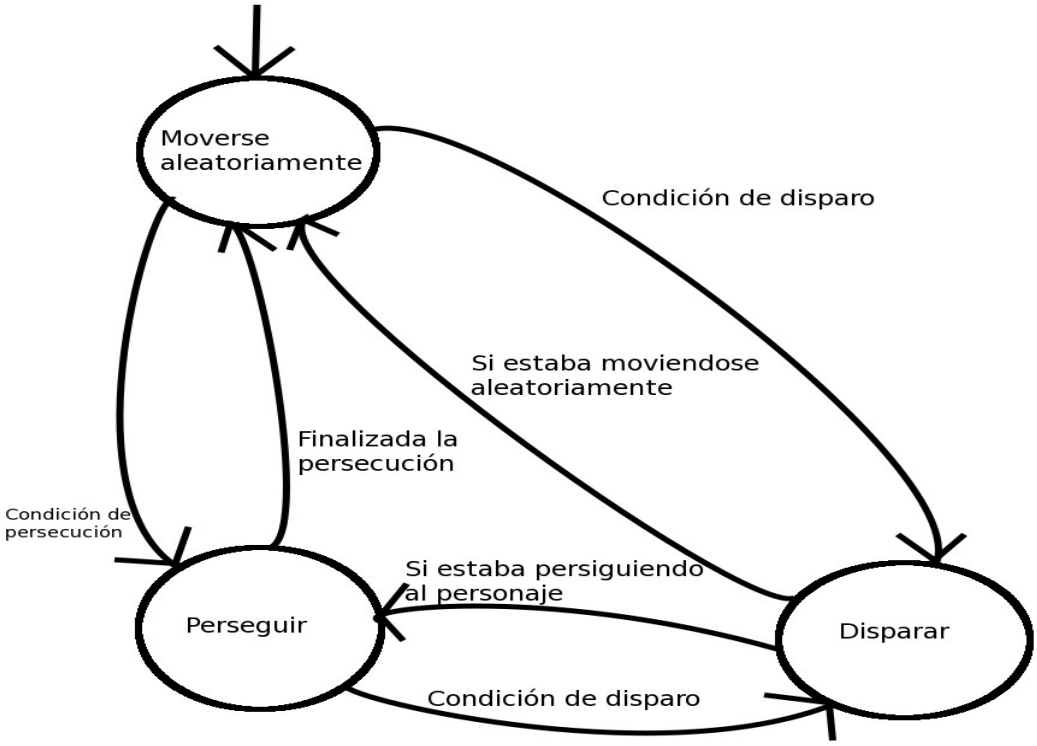


Figura 10.- Máquina de estados finitos correspondiente al comportamiento del enemigo principal de la historia.

Esta máquina muestra los tres estados principales en los que se encuentra el enemigo cuando su acción es disparada en la escena y hasta que muere, no obstante esta máquina esta compactada, dado que si extendemos las condiciones de transición, que dependiendo de la salud del enemigo, hacen que sea una transición mas o menos probable(aumenta la agresividad al perder salud), tendríamos que en cualquier momento se puede transitar a un grupo de tres estados iguales pero con transiciones mas probables al llegar a ciertos umbrales de salud.

Esto se da dos veces, al bajar la salud a la mitad y cuando llega a muy baja salud, donde la probabilidad de disparar por ejemplo en cualquiera de los dos otros estados es bastante mas alta.

La condición de transición al estado “Disparar” es la misma tanto en el estado “Moviéndose aleatoria mente”, como en el estado “Persiguiendo al personaje”, y siempre se pasa automáticamente de vuelta al estado del que se proviene después de haber disparado.

Ambas transiciones(a “disparar” y a “perseguir”) estan sujetas a probabilidades, dependientes de la salud del enemigo como ya se ha dicho, pero no obstante con un mínimo de tiempo para prevenir comportamientos nerviosos que es mas pequeño en cada umbral de salud, y que finalmente se calcula de forma aleatoria pasado este tiempo.

Finalmente, la transición de vuelta al estado “Moverse aleatoria mente” desde “Perseguir” es simplemente disparada cuando pasa un tiempo de forma determinista.

Figura 11.- Acción enemiga. El antagonista se mueve a tu alrededor persiguiéndote, mientras te lanza su magia, que te resta salud.

Módulo de eventos y control de flujo de la aplicación: Este módulo es el control centralizado de la capa de lógica de la aplicación, que se comunica con los otros módulos de la misma.

Esta formado por dos partes, la primera parte es un control de flujo gobernado por una FSM y la segunda es un control de variables globales y de ciertos aspectos de la aplicación que funciona de forma global, sin tener en cuenta el momento o estado de la historia en el juego.

Se ha diseñado con una estructura que aunque lineal, te permite moverte con libertad y realizar varias acciones en distintas situaciones en el desarrollo de la acción.

La FSM que controla el flujo de la acción en la aplicación es la que sigue:

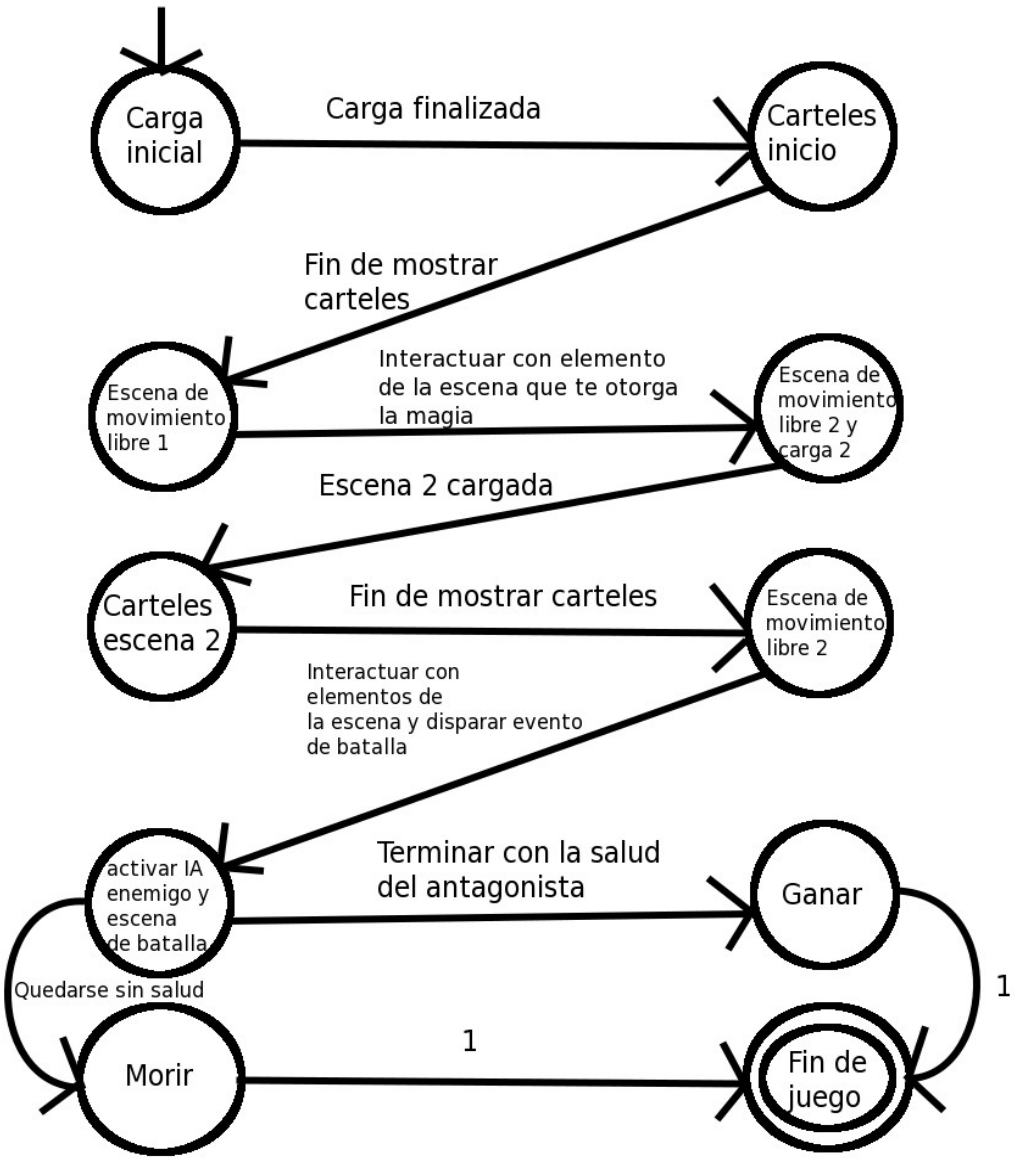


Figura 12.- Máquina de estados finitos que controla el flujo de la aplicación y sus estados.

Como se puede apreciar se empieza en la pantalla de carga, y se pasa a un estado en la primera escena de las Llanuras mediante la transición de fin de carga, en la cual se muestran carteles que funcionan a modo de tutorial, explicando al usuario lo que debe hacer para moverse e interactuar con el escenario.

El fin de estos primeros carteles te llevan a la primera escena libre, en la cual puedes moverte por esta escena e interactuar con los objetos.

Sucesivamente, si disparas las transiciones de los estados realizando las acciones que las desencadenan, esta FSM te guía por el transcurso de la acción y la historia del juego, de modo que controla por ejemplo tu posición y la de la cámara, si te puedes mover, si el enemigo está activo, etc.

Finalmente, después de quedarte sin salud o de que el enemigo se quede sin salud en la escena final, se accede a un estado en el que el personaje muere o alcanzas la victoria, estados en los que se realizan las acciones necesarias para que esto suceda y para hacer la limpieza necesaria(por ejemplo, desactivar la IA del antagonista). Tras estas acciones se accede inmediatamente por cualquiera de los dos caminos al estado de juego final, en el cual no se realiza ninguna acción adicional(la música sigue sonando, el cartel de victoria o muerte sigue apareciendo, pero no se realiza ni puedes realizar ninguna acción mas).

3.2. Modelado de modelos 3D, animaciones, creación de texturas y generación de las piezas de audio.

3.2.1. Modelos 3D.

La aplicación consta de ciertos modelos 3D que, en su mayor parte han sido descargados y utilizados a partir de las siguientes páginas web de recursos:

http://www.turbosquid.com/

https://clara.io

https://www.cgtrader.com

No obstante, se han cargado y editado con diferentes herramientas, las cuales son:

- Blender, el programa libre y de código abierto que permite crear modelos y animaciones en 3D.
- 3D Studio Max, el programa de Autodesk que permite también modelar y animar en 3D.
- Script para leer archivos .obj y convertirlos al formato JSON que lee Three.js, procedente de las utilidades creadas conjuntamente con la librería.

El formato JSON es el que se ha utilizado en la carga de los modelos en la aplicación mediante la funcionalidad JSONLoader de Three, este es un formato en el que se colocan las diferentes partes de la información referente al modelo, sus materiales, animaciones, etc. de forma estructurada.

En Blender, se ha utilizado el exportador io-three que se proporciona al descargar la librería Three.js. Cabe decir que dicho exportador no parece funcionar del todo correctamente aun, pues no exporta modelos con varias mallas sino que solo exporta una.

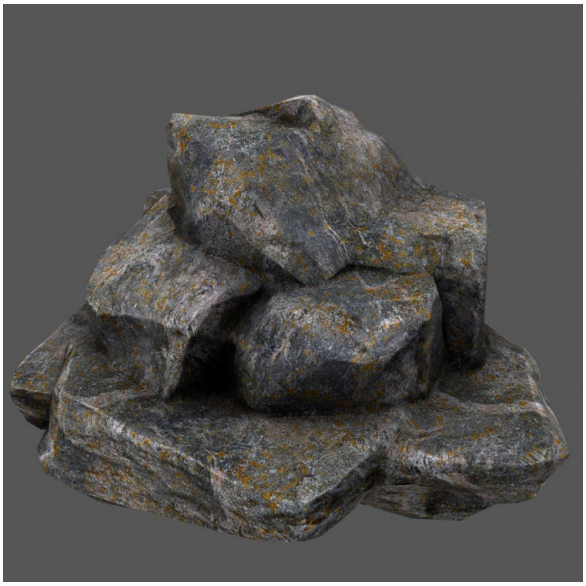
En 3D Studio Max, se ha utilizado el conversor a .obj para luego convertir de este formato a JSON mediante el script de utilidades de conversión que se proporciona con Three.

Estos son algunos de los modelos que se han utilizado en el desarrollo de la aplicación y se han importado mediante el JSONLoader en ella:

Figura 13.- Modelo 3D del personaje de la historia.

Figura 14.- Modelo 3D del antagonista.

Aquí se pueden ver ciertos modelos de objetos de escenografía:



Figuras 15 y 16.- Modelo de una roca y de un árbol, respectivamente.

Algunos de los modelos 3D se han generado mediante funciones de la propia librería Three.js, como por ejemplo el suelo que es un plano, o los edificios de la segunda escena que son rectangulares.

3.2.2. Animaciones de los modelos.

La mayoría de animaciones se generan en la propia aplicación mediante Three, no obstante se ha creado una animación mediante la herramienta Blender para el modelo del enemigo o antagonista, creando primero el esqueleto, y luego utilizándolo para hacer el *skinning* al modelo(asignar grupos de polígonos a los huesos), para mas tarde finalmente, crear la animación y exportar a formato JSON el modelo con la información relativa al esqueleto y la propia animación.

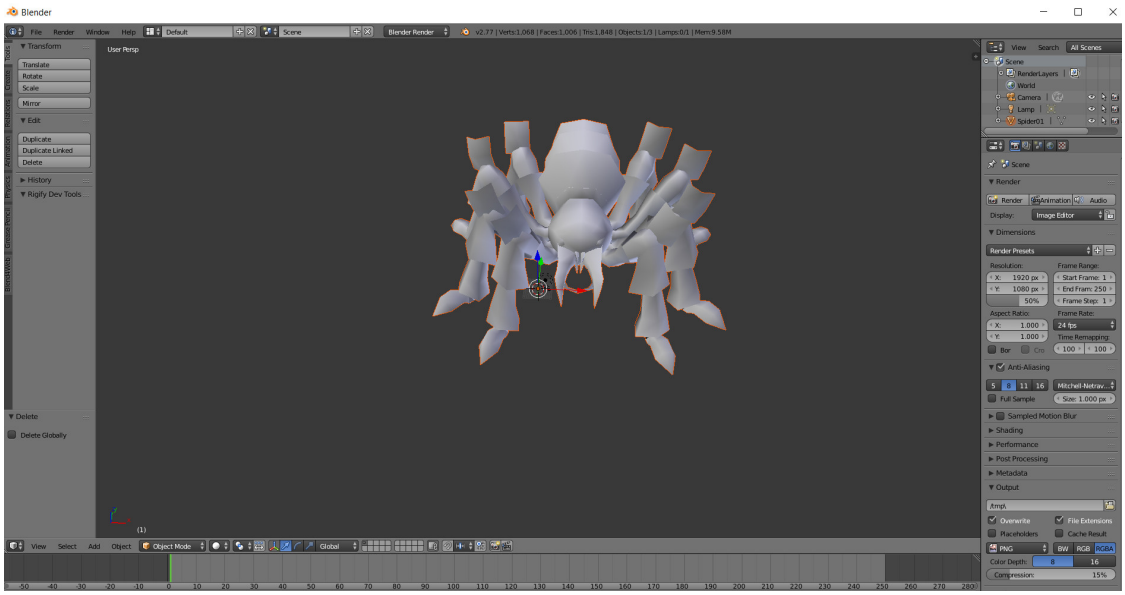


Figura 17.- Modelo 3D de la araña que hace de enemigo sin esqueleto ni animaciones.

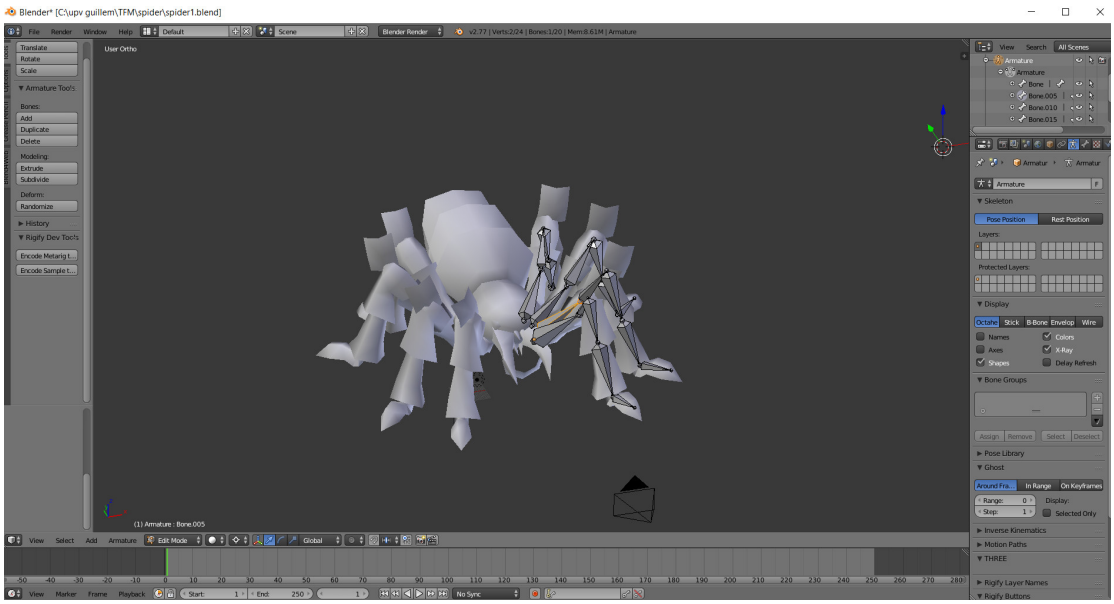


Figura 18.- Modelo 3D de la araña que hace de enemigo con un inicio de esqueleto.

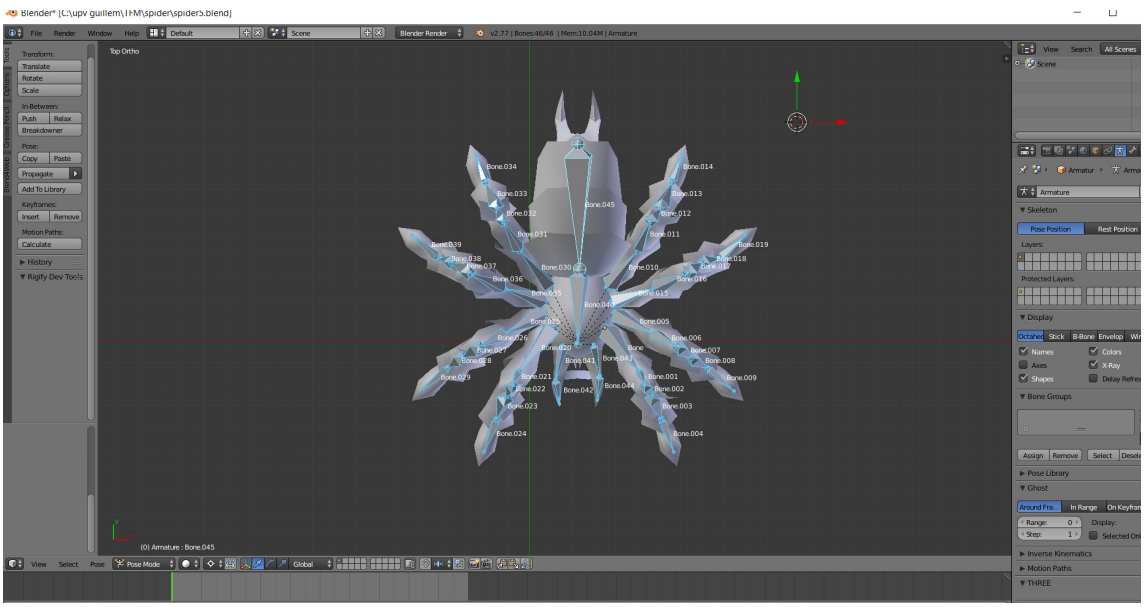


Figura 19.- Modelo 3D de la araña que hace de enemigo con el esqueleto completo y con el *skinning* realizado, de modo que está lista para ser animada.

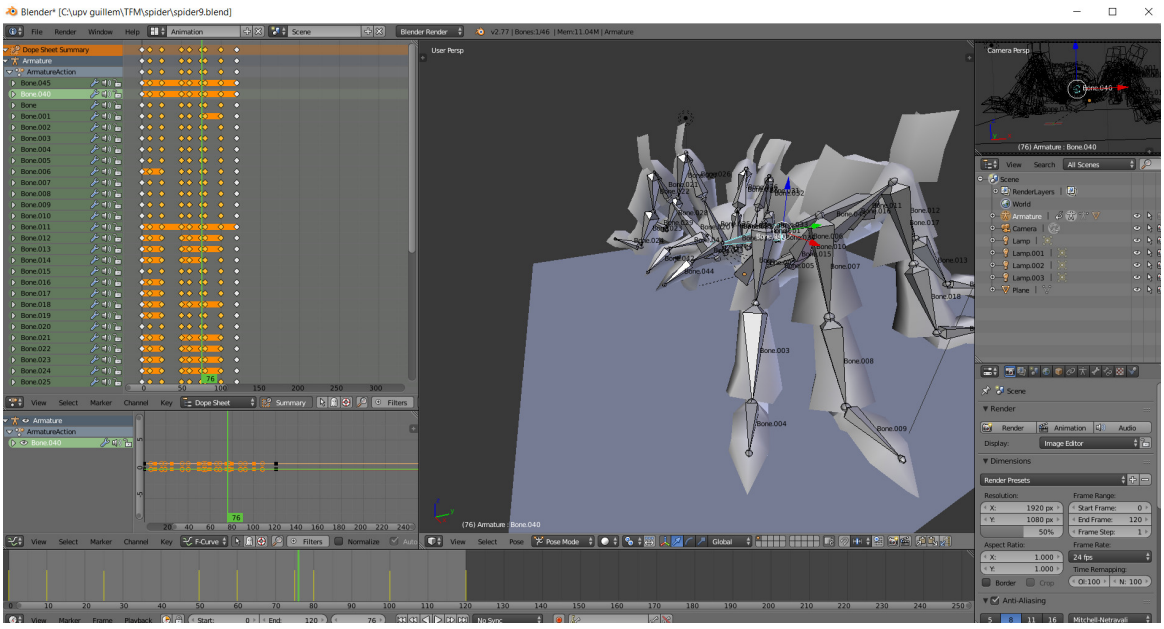


Figura 20.- Modelo 3D de la araña que hace de enemigo con la animación ya realizada, se puede apreciar la ventana de Animación de Blender con todos los fotogramas clave de cada hueso.

Tras realizar la animación y exportarla, ya es tarea del cargador JSONLoader el cargar la el modelo y asignarlo a una Sietemesino, que permitirá poseer una animación de esqueleto o ArmatureAction, y posteriormente esta acción se ejecutará mediante un AnimationMixer.

3.2.3. Generación de texturas en GIMP.

El programa GIMP es un software de manipulación de imágenes libre que permite que el usuario cree y manipule imágenes con varias capas, textos, colores, formas, etc. Posee diversas herramientas de selección de áreas, de pintura, de transformaciones o de color.

Se han creado texturas en GIMP que luego serían usadas en la aplicación como cajas de texto o *TextBoxes*, estas texturas se crean a partir de una base vacía de caja de texto descargada de la red:

Figura 21.- Base para los *TextBox* creados durante el desarrollo del juego.

Se editan con GIMP para darles el aspecto deseado que tendrá la caja de texto en la aplicación finalmente, y se cargan luego con la función loadTexture del módulo ImageUtils de Three.js:

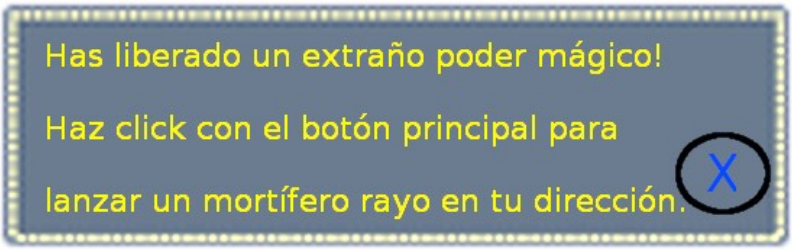


Figura 22.- Textura para una de las cajas de texto del juego, se puede apreciar un efecto de manipulación de brillo y contraste para que tenga mejor visibilidad dentro de la aplicación.

3.2.4. Edición de piezas de audio y efectos sonoros.

Para la edición de piezas de audio y efectos, se ha utilizado el programa de libre acceso *Audacity*. Con este programa, se ha podido recortar piezas de audio para que se ajusten a las necesidades de la aplicación, además de amplificar efectos que no se oían bien dentro del entorno del juego.

Se han utilizado tres temas musicales distintos, además de un tono de victoria y otro de derrota que se reproducen en bucle, y algunos efectos sonoros, como un efecto de monstruo muriendo, por ejemplo, para el momento en que el jugador derrota al antagonista.

3.3. Implementación del código de la aplicación.

3.3.1. Estructura del código.

La aplicación cuenta con un documento .html que contiene la información de la estructura de la página, la cual es sencilla, dado que simplemente se crea una página con un Canvas que luego podrá utilizar WebGL para dibujar gráficos en él.

También se cargan todos los scripts JavaScript necesarios de librerías y de código implementado de los módulos de la aplicación.

Estas librerías utilizadas son:

- El propio Three.js.

- OrbitControls.js y dat.gui.min.js que son, respectivamente, una librería que crea un efecto con el que se puede orbitar en la escena con el ratón y una librería para crear GUIs(Graphical User Interface o interfaces de usuario, como widgets con controles de variables, por ejemplo). Se han utilizado únicamente con motivos de depurar código, y en la implementación final no se utilizan finalmente.

- Tween.js, que es una librería que permite tener una interfaz con la que crear un efecto de pasado de un estado a otro(*tweening*) de propiedades numéricas de objetos y propiedades de estilo CSS, sin embargo solo se han utilizado las primeras.

- Los módulos de la aplicación desarrollados.

A continuación se procederá a explicar con mas detalle la implementación de los módulos de la aplicación.

3.3.2. Implementación de los módulos de la aplicación.

3.3.2.1. Capa de presentación gráfica.

La capa de presentación gráfica consta del código necesario para mostrar los gráficos y efectos, además de música y efectos sonoros.

Podemos decir que estos elementos gráficos se componen de mallas, efectos generados a partir de estas, como las sombras arrojadas, partículas(*sprites* generados con un gradiente radial), algunos elementos gráficos sencillos con un material como son la barra de salud o energía del HUD, y elementos de audio que emiten música y efectos en la escena.

Todos estos elementos se tienen que crear y añadir al grafo de escena para poder ser renderizados o generados al cargarse la misma, la mayor parte se crean en la función loadScene que carga la propia escena al iniciarse la aplicación.

3.3.2.2. Capa de lógica.

Como ya se ha explicado, esta capa consta de varios módulos con control centralizado en el módulo de eventos y control de flujo, en adelante se detalla su implementación:

Módulo de físicas: El módulo de físicas tiene varias funciones o partes que lo componen, a saber:

1. Detección de colisiones:

- Funciones de detección de colisiones entre cajas de inclusión:

Es la parte esencial y mas básica, de mas bajo nivel que componen la detección de colisiones, y consta de dos funciones, una que calcula colisiones entre cajas de inclusión circulares y otra entre cajas de inclusión circulares contra otras cuadradas.

Lo que hacen estas funciones es determinar si una caja de inclusión tiene cierta superposición con otra, esto es, que la intersección no es nula.

```
function circleAndRectCollided(circle_x, circle_z, circle_r, rect_x, rect_z, rect_w, rect_h) {
   var circleDistanceX = Math.abs(circle_z - rect_z - rect_w/2);
   var circleDistanceY = Math.abs(circle_x - rect_x - rect_h/2);

   if (circleDistanceX > (rect_w/2 + circle_r)) { return false; }
   if (circleDistanceY > (rect_h/2 + circle_r)) { return false; }

   if (circleDistanceX <= (rect_w/2)) { return true; }
   if (circleDistanceY <= (rect_h/2)) { return true; }

   var cornerDistance_sq = Math.pow(circleDistanceX - rect_w/2, 2) + Math.pow(circleDistanceY - rect_h/2, 2);

   return (cornerDistance_sq <= (Math.pow(circle_r, 2)));
}

function circleAndCircleCollided(circle_x, circle_z, circle_r, other_x, other_z, other_r){

    return Math.sqrt(Math.pow(circle_x - other_x, 2) + Math.pow(circle_z - other_z, 2)) <= (circle_r + other_r);

}
```

Figura 23.- Código de las funciones que detectan superposición entre cajas de inclusión, módulo de físicas, parte de colisiones.

- Funciones que calculan las colisiones en general en una escena dada:

Hay dos funciones, una por cada escena, y cada una calcula las colisiones de una u otra escena mediante el uso de las funciones básicas de superposición de cajas de inclusión ya vistas.

Estas funciones simplemente llaman a las funciones anteriores una vez por cada objeto colisionable de la escena, incluido el *skyBox* o caja de entorno, que además requiere el uso de cuatro llamadas(una por cada borde de la escena).

También hay otra escena que tiene en cuenta la presencia del enemigo principal, de modo que calcula también las colisiones referentes a este elemento dinámico de la escena, tanto con el protagonista como con los demás elementos.

```
function collisionLlanuraNoMob(x, y, z, circleRadius){

    //Skybox
    if(circleAndRectCollided(x,z,circleRadius,5000+sceneDistanceX,-5500+sceneDistanceZ,11000,100)) return true;
    if(circleAndRectCollided(x,z,circleRadius,-5500+sceneDistanceX,-5100+sceneDistanceZ,100,11000)) return true;
    if(circleAndRectCollided(x,z,circleRadius,-5500+sceneDistanceX,5000+sceneDistanceZ,100,11000)) return true;
    if(circleAndRectCollided(x,z,circleRadius,-5100+sceneDistanceX,-5500+sceneDistanceZ,11000,100)) return true;
    //Rocas
    if(meshRock) if(circleAndCircleCollided(x,z,circleRadius,meshRock.position.x,meshRock.position.z,300)) return true;
    /*if(circleAndCircleCollided(x,z,circleRadius,rock1.position.x,rock1.position.z,rock1.geometry.parameters.radius-2)) return t
    //arboles

    if(meshTree) if(circleAndCircleCollided(x, z, circleRadius,meshTree.position.x-55, meshTree.position.z+125, 20)) return true;
    //if(meshTree2) if(circleAndCircleCollided(x, z, circleRadius,meshTree2.position.x, meshTree2.position.z, 30)) return true;

    if(meshTreeMax) if(circleAndCircleCollided(x, z, circleRadius,meshTreeMax.position.x, meshTreeMax.position.z, 30)) return true
    if(meshTreeMax2) if(circleAndCircleCollided(x, z, circleRadius,meshTreeMax2.position.x, meshTreeMax2.position.z, 30)) return t
    if(meshTreeMax3) if(circleAndCircleCollided(x, z, circleRadius,meshTreeMax3.position.x, meshTreeMax3.position.z, 30)) return t

    //pilares
    if(meshOrb) if(circleAndCircleCollided(x, z, circleRadius,meshOrb.position.x, meshOrb.position.z, 165)) return true;
    if(meshOrb2) if(circleAndCircleCollided(x, z, circleRadius,meshOrb2.position.x, meshOrb2.position.z, 165)) return true;
    return false;

}
```

Figura 24.- Código de la función que calcula las colisiones para una entidad de la cual proporcionamos su posición y radio de caja circular de inclusión en la primera escena.

Mas tarde estas funciones son llamadas en la función de actualización principal de la aplicación o función *update*, la cual llama a la función de inteligencia artificial del enemigo y esta a su vez a una de estas funciones también para detectar colisiones del enemigo. Todo esto controlado a su vez por el módulo de control centralizado.

2. Balística:

- Generación de trayectorias e impactos:

Para poder trazar una trayectoria en el espacio a partir de un punto dado, se utiliza la clase *Raycaster* de Three.js.

Dicha clase permite trazar rayos en el espacio desde un punto en el mismo y con una dirección. A partir de esta base, lo que la aplicación hace es generar rayos cada vez que se dispara, desde la posición del usuario del rayo, para comprobar si hay alguna colisión con algún objeto colisionable(hay objetos como las partículas que no son colisionables por el rayo, la intersección con ellos se ignora). En el siguiente *frame*, si no hubo una intersección con un objeto válido y no se ha llegado al alcance del arma, se coloca otro rayo en el final del primero y con su misma dirección(sustituye al inicial), calculando este final como el punto inicial mas el alcance del rayo. De este modo se sigue hasta alcanzar un objeto y llamar al resolvedor de colisiones correspondientes o terminar el alcance del rayo sin colisionar con nada.

- Dibujado de los rayos emitidos en la escena:

Los rayos emitidos deben ser dibujados para proveer al usuario de una realimentación visual que le indique que efectivamente ha disparado un rayo, esto se hace en dos pasos:

Primero: se crean un cierto número de partículas definidas como objetos THREE.Sprite, que se colocarán en la escena en el lugar que corresponda, por ejemplo ciertas partículas deben ir al lado del personaje cuando la magia esté activa.

Segundo: se establecen parámetros de *tweening* sobre propiedades como la posición y la escala de estas partículas y se introducen en la escena.

Tercero: se utiliza el módulo de eventos y control de flujo para controlar estas partículas, de modo que solo se renderizen y se calculen las propiedades cuando sea necesario y hagan las animaciones pertinentes como de rayo, efecto expansivo de colisión o simplemente reposo animado al lado del personaje.

2. Resolvedores de colisiones:

- Resolución de colisiones físicas entre objetos tangibles:

Cuando dos objetos físicos tangibles(como una roca o el personaje) colisionan entre ellos, se re calcula la posición del nuevo frame del objeto colisionante para que no penetre el objeto colisionado, y se utiliza esta en vez de la primera nueva posición calculada por la función de actualización de movimiento de los objetos dinámicos, tanto en la propia función *update* para el personaje, como en la función correspondiente a la inteligencia artificial del antagonista, que calcula sus actualizaciones.

- Resolución de colisiones de rayos con objetos físicos(intersección del *Raycaster*):

La intersección del *Raycaster* con un objeto alcanzable por el rayo llama a una de las dos funciones de resolución de colisiones para estos casos(una para el enemigo y otra para el personaje), que dependiendo del objeto que hayan alcanzado realizan una acción u otra, y además colocan una animación de onda expansiva de partículas y reproducen un sonido en el lugar de colisión. También se reproduce un sonido al disparar.

Módulo de carga: El módulo de carga realiza acciones de cargado de modelos 3D, audio, texturas y demás elementos que serán introducidos en la escena.

Su ubicación es la función *loadScene*, llamada al iniciarse la aplicación, que carga texturas mediante la función loadTexture de imageUtils dentro de Three.js, carga modelos mediante un JSONLoader, que ejecuta un *callback*(función que se pasa como parámetro para ser llamada luego) cuando estos se cargan, para introducirlos en la escena, animarlos o realizar otras acciones, y carga audio mediante un audioLoader que también ejecuta un *callback* cuando el audio se ha cargado.

```
// load a resource
    audioLoader.load(
    // resource URL
    'achievementUp.ogg',
    // Function when resource is loaded
    function ( audioBuffer ) {

        // set the audio object buffer to the loaded object
        effectsAchievement.setBuffer( audioBuffer );

        chargue += 1;

    },
    // Function called when download progresses
    function ( xhr ) {
        //console.log( (xhr.loaded / xhr.total * 100) + '% loaded' );
    },
    // Function called when download errors
    function ( xhr ) {
        //console.log( 'An error happened' );
    }
);
```

Figura 25.- Código que carga un efecto sonoro de consecución de un objetivo.

Módulo de captura de eventos: El módulo de captura de eventos se encarga de recoger la interacción del usuario. Su implementación utiliza los eventos de usuario:

- *Mousemove*: Este evento de usuario lanza una función que mueve la cámara de un cierto modo controlado a ambos lados, 360 grados.
  Esta forma de mover la cámara se trata de un control en el cual, al tener el mouse en el centro de la escena no hay rotación, pero al desplazarnos un poco a derecha o izquierda se empieza a rotar cada vez mas hacia ese lado(mas conforme nos acercamos al borde de la pantalla), pero teniendo en cuenta que al llegar el mouse a los bordes de la pantalla(tanto superiores como inferiores), se deja de rotar, para no marear al usuario.
  Funciona también con el *touchpad*.

- *Click*: Su función es, dependiendo de varias variables como si hemos conseguido o no los poderes mágicos, o si tenemos energía para lanzar un rayo, lanzarlo en nuestra dirección.
  Cabe decir que el usuario solo puede disparar un rayo a la vez y que este esta limitado por la cantidad de energía que le reste, la cual disminuye cada vez que dispara. Sin embargo, el enemigo principal puede lanzar rayos sin ninguna de estas restricciones.
  También funciona con el *touchpad*.

- *Keydown*: Detecta ciertas teclas al ser pulsadas por el usuario, para que así pueda interactuar con el teclado con la aplicación.
  Estas teclas son:

  - Tecla M: Activa o desactiva los poderes mágicos cuando ya han sido conseguidos.

  - Tecla X: Cierra carteles que aparecen como parte del HUD que se pueden cerrar y se indiquen como tal.

  - Tecla Z: Interactuar con el entorno.

  - Tecla W y tecla S: Respectivamente, aumenta o reduce la aceleración del usuario cuando este puede moverse. Esta aceleración viene dada por una función exponencial, de modo que al inicio acelera despacio pero muy pronto llega a su tope de velocidad, y si se suelta baja muy rápido la aceleración. Esto se ha realizado de este modo debido a que es un comportamiento bastante cercano a la forma de acelerar de los seres humanos.
    Se reduce ligeramente la aceleración con el tiempo si nos estamos moviendo, también hay aceleración hacia atrás para simular andar hacia atrás, y esto también se reduce(aumenta ligeramente dado que es negativa) con el tiempo si estamos en movimiento.

- *Resize*: Actualiza el tamaño de la aplicación dependiendo de si cambia el tamaño de la ventana y mantiene la relación de aspecto.

Módulo de control de cámara: Este módulo corresponde a la implementación de captura de evento de *Mousemove* ya mencionada y también al OrbitControls durante depuración, que permite mas movimientos como *zoom*, *pan*, etc.

Módulo de animaciones: Este módulo se compone de ciertas funciones con las que se animan personajes, que son llamadas en el *callback* en que se cargan sus modelos o en otras partes cuando sea necesario, no obstante, finalmente solo se utilizó la animación del enemigo principal.

```
function animate2(spider) {

    mixerSpider = new THREE.AnimationMixer( spider );

    var materials = spider.material.materials;

    for (var k in materials) {
        materials[k].skinning = true;

    }


    var bonesClip = spider.geometry.animations[0];



    actionS = mixerSpider.clipAction( bonesClip, spider );

    actionS.play();

}
```

Figura 26.- Código que anima al antagonista.

Se han utilizado *AnimationMixer* para las animaciones, dado que es la funcionalidad de Three que mejor funcionó en el proyecto, tras probar otros como los M*orphTargets*.

También existen otros movimientos animados en la escena como el del enemigo hacia su dirección y su rotación que son decididos y ejecutados por su inteligencia artificial.

Para actualizar las animaciones se utiliza un THREE.Clock y el *DeltaTime*, que no es otra cosa que el tiempo transcurrido entre una llamada y otra a la función que da este tiempo, en la función de *render* de la aplicación.

Módulo del HUD: Este módulo es el encargado de mostrar ciertos elementos en la escena, que acompañan al usuario, como son las barras de energía(color verde), salud(color rojo) y los carteles que aparecen en ciertas ocasiones anunciando por ejemplo la carga de la escena o que se ha conseguido los poderes mágicos.

Se decidió crear elementos dentro de la aplicación y no utilizar *div* distintos de la página web para estos elementos, ya que se pretendía que toda la aplicación estuviese embebida dentro del mismo Canvas. De este modo puede ser mostrada toda la aplicación en otro lugar utilizando un solo elemento de la página web, sin tener que arrastrar varios.

Inicialmente se crean los carteles y las barras al crearse el jugador. Mas tarde en la función de actualización principal se llama a un par de funciones para actualizarlos, una para la rotación y otra para la posición.

Finalmente, en la función correspondiente al módulo de eventos que aun no se ha explicado, se hacen o no visibles estos elementos dependiendo de la situación.

Módulo de comportamiento enemigo: También llamado Inteligencia Artificial del rival, este módulo consta de varios elementos, la FSM que gobierna el comportamiento del enemigo mas el cálculo de su movimiento, su detección de colisiones, la actualización de su rayo en caso de que haya disparado y otras actualizaciones necesarias de sus características.

```
function spiderIa(){

    //FSM del Antagonista

    //Nos guardamos la rotacion
    var rotBack = spider.rotation.y;
```

Figura 27.- Inicio de la función de la inteligencia artificial del enemigo principal.

Módulo de eventos y control de flujo: Este módulo como se ha explicado es el control central de la capa de lógica, que interactúa con los demás módulos.

Tiene dos elementos:

- Función de control cinemático de la escena: Controla el flujo de la acción mediante una FSM ya expuesta en apartados anteriores.

- Actualización de variables y elementos globales: A parte de esta función de control de flujo que es llamada en cada llamada al propio módulo en la función de actualización principal de la aplicación, se realizan ciertas acciones globales también como mostrar o no una división de la barra de salud, por ejemplo. O también actualizar la salud del personaje si el rival y él han colisionado de alguna forma.

```
function motorEventos(){

    //Control de flujo cinematico
    cinematicFluxControl();

    /*if(cargado == true && false){

    if(spiderCinematicAction == true){
         if(colissionEnemy==true) {
            if(hitInterval==maxHitInterval) {
                colissionEnemy = false;
                lifePj -=5;
                hitInterval = 0;
            } else { hitInterval++; colissionEnemy=false;}
         }
         for(y=0;y<100;y++){

             if(lifePj > y) lifebar[y].visible = true;
             else lifebar[y].visible = false;

         }
    }


    if(iniciadoEnLaMagia){

        //Dibujar barra de magia y actualizar
        if(magicNow<100){
            magicCount +=1;
```

Figura 28.- Fragmento de la función correspondiente al módulo de control central, que a su vez llama a su función auxiliar de control de flujo de acción y realiza otras acciones globales.

Parte 4

Experimentación realizada

4.1. Preparación del experimento.

Para la realización del experimento, se han diseñado 4 casos de prueba:

- Google Chrome bajo Windows.
- Mozilla Firefox bajo Windows.
- Google Chrome bajo Linux(Ubuntu).
- Mozilla Firefox bajo Linux(Ubuntu).

Se pretendía medir el rendimiento de la aplicación en los distintos casos de prueba preparados, mas la fluidez y la usabilidad de la misma, con ayuda de datos numéricos como el ratio de FPS(*frames per second* o *frames* por segundo) y el tiempo de carga inicial de la aplicación.

Además, como se ha percibido que la aplicación necesita mas tiempo y tiene menos rendimiento la primera vez que es cargada en un navegador dado, comparado con sucesivas cargas, en las que se utilizan datos en caché para ahorrar tiempo, se han medido datos en la primera y en una carga ya realizada tras cierto tiempo de actividad en los cuatro casos de prueba, para enriquecer la información que arrojen los experimentos.

Como es sabido ademas que WebGL trabaja con la tarjeta gráfica con que esté equipado el computador del cliente, se han realizado todos los experimentos en la misma máquina para proporcionar la mayor veracidad posible a los resultados.

Se ha creado una versión del código aparte, partiendo de la inicial, que imprime una gráfica del ratio FPS en todo momento, y que además calcula el tiempo que se tarda en cargar la escena en la carga inicial y lo muestra por pantalla, de modo que así se pueda realizar los experimentos cómodamente y de forma cuantitativa.

4.2. Realización del experimento y resultados obtenidos.

Se procederá a detallar en este apartado los cuatro casos de prueba sujetos a test y sus resultados, para posteriormente utilizarlos para sacar las conclusiones pertinentes:

- Caso de prueba 1, Google Chrome bajo Windows:

En la carga inicial, se ha obtenido un tiempo de carga de 44 segundos.

El ratio de FPS se ha mantenido alto en 60(límite impuesto) aproximadamente durante toda la primera parte de muestra de carteles e información al usuario y de libre movimiento sin elementos adicionales por la escena.

Al liberar las partículas en la primera escena(poder mágico), se ha perdido un poco de rendimiento, pero muy poco(se ha bajado a unos 58 FPS), y finalmente en la escena con mas carga del juego(enemigo activo y atacando, personaje activo y atacándole, varios sistemas de partículas), se ha bajado a un ratio de entre 25 y 44 *frames* por segundo.

Podemos decir que la fluidez y el rendimiento han sido elevados en este caso de prueba.

En cuanto a la usabilidad, podemos decir que ha habido un movimiento fluido por las escenas, y aunque se puede llegar a cargar un poco en el momento de máxima carga computacional, la usabilidad ha resultado bastante buena.

En el caso de la carga realizada después de un tiempo de juego, se ha obtenido un tiempo de carga inicial de 7,9 segundos.

El ratio FPS se ha mantenido muy similar al anterior, cambiando a ser algo mas alto en el momento de máxima carga(entre 35 y 53 FPS).

- Caso de prueba 2, Mozilla Firefox bajo Windows:

En la carga inicial se ha observado un tiempo de 49,2 segundos.

Respecto al ratio de FPS, se ha observado que al inicio se tiene un ratio primero de 55 y luego conforme el personaje se mueve baja a estar entre 30 y 47 FPS, de modo que va fluido pero a veces se carga mucho y da pequeños saltos.

Con el sistema de partículas baja a estar entre 23 y 42 FPS, y luego en la escena de mayor carga baja a estar un poco por debajo de 20 de media.

Se puede concluir que aunque funciona de forma fluida la mayor parte del tiempo, da pequeños saltos en ocasiones y además el ratio de FPS en general es mas bajo que en el caso de pruebas de Chrome bajo el mismo sistema operativo.

Respecto a la usabilidad, podemos decir que es aceptable, pero no excesivamente buena, dado que los pequeños saltos o momentos de sobrecarga pueden afectar a usuarios impacientes, aunque si bien es cierto que si se tiene un *hardware* muy potente el efecto tiende a disminuir.

Con la carga realizada después de tener la aplicación un tiempo cargada, el tiempo de carga inicial disminuye a 9,3 segundos, y el ratio de FPS aumenta ligeramente, con lo cual la fluidez también es mayor y es bastante aceptable, excepto en el momento de mayor carga que sobrecarga un poco al navegador.

Estos pequeños saltos también se ven mitigados por este hecho de tener la aplicación ya un tiempo cargada, de modo que su efecto se desvanece progresivamente en sucesivas cargas.

- Caso de prueba 3, Google Chrome bajo Linux:

En la carga inicial, se ha observado un tiempo de carga de 44 segundos.

El ratio de FPS ha empezado en mas de 50, bajando luego a estar entre 38 y 42 al empezar a moverse, para pasar luego a estar entre 26 y 33 en el momento en que se consiguen y usan partículas.

Finalmente baja a estar entre 17 y 30 con la mayor carga.

Podemos concluir que la fluidez y rendimiento no son malos, son aceptables, pero desde luego son menores que los observados con el mismo navegador bajo Windows.

Respecto a la usabilidad, podemos decir que es aceptable pero no tan buena como la percibida bajo Windows, y que esto afecta directamente al usuario.

Ya con la aplicación cargada un tiempo y en sucesivas cargas observamos que el tiempo inicial de carga baja a 9 segundos, y que el ratio FPS de nuevo aumenta ligeramente en general.

- Caso de prueba 4, Mozilla Firefox bajo Linux:

En la carga inicial se ha observado un tiempo de 54 segundos.

Su ratio de FPS ha empezado en 35 y bajando a 25 de media al empezar a interactuar con la escena, además luego ha seguido bajando a estar en unos 20 de media con las partículas en escena y finalmente ha bajado a estar entre 10 y 23 FPS en la escena final.

Aunque luego ha mejorado en la carga después de tener la aplicación un tiempo a tener cierta fluidez(ha aumentado algo el ratio de FPS), el rendimiento ha sido entre bajo y normal en este caso, así como la fluidez.

Se observan además incluso parones notables en ocasiones, sobretodo en la escena de máxima carga en la primera carga de la aplicación, aunque luego en sucesivas cargas son pequeños parones.

Respecto a la usabilidad, podemos concluir que con este rendimiento y forma de llevar la aplicación, no resulta excesivamente usable en Mozilla Firefox bajo Linux, siendo el caso de prueba con peores resultados obtenidos.

El tiempo de carga inicial ha bajado a 11,5 segundos en cargas sucesivas de la aplicación.

4.3. Conclusiones extraídas.

Podemos sacar varias conclusiones de estos datos:

- La fluidez, la usabilidad y el rendimiento de la aplicación, se ha observado que son mejores en el navegador Chrome de Google que en el navegador Firefox de Mozilla, dado que los ratios de FPS y fluidez generales lo demuestran en los casos de prueba de ambos sistemas operativos, Windows y Linux. Esto puede tener diversas causas, como que el navegador de Mozilla no tenga tantas optimizaciones como Chrome, por ejemplo.

- Se ha observado un mejor rendimiento bajo Windows que bajo Linux, en todos los casos de prueba. Esto podría deberse a que los drivers del sistema operativo Windows aprovechen mejor la potencia de la tarjeta gráfica, que es utilizada por WebGL. También podría deberse a posibles optimizaciones realizadas para los navegadores para el caso de cuando se ejecuten bajo Windows que no existan para cuando se ejecuten bajo Linux.

- Uno de los cuellos de botella principales del rendimiento de la aplicación ha resultado ser la carga de los elementos desde el servidor, ya que ha sido el momento en que mas tiempo ha estado cargando la aplicación en todos los casos de prueba y además ha disminuido considerablemente en sucesivas cargas, ya que al estar este contenido ya descargado y almacenado en la *cache* del navegador, no era necesario traerlo desde el servidor.
  Hay que tener en cuenta que el servidor de la Universidad Politécnica tiene limitaciones, y traer elementos de audio, texturas y otros archivos desde allí puede resultar costoso.

- Una posible optimización seria mejorar o bien la carga desde servidor o reducir el peso de los archivos multimedia que usa la aplicación. Las partículas también afectan a la fluidez.

- En general el rendimiento, fluidez y usabilidad de la aplicación son buenos, pero hay casos en que no lo son tanto y se producen pequeños parones, aunque es cierto que la potencia del *hardware* del cliente también influye, como ya se ha indicado anteriormente.

Parte 5

Conclusiones y trabajo futuro

5.1. Conclusiones finales.

Debido al gran número de expectativas creadas con esta nueva tecnología, que permite renderizar gráficos 3D interactivos en el navegador sin plug-ins, la comunidad internacional se ha volcado en tratar de crear librerías y funcionalidades conjuntamente con las herramientas existentes de modelado, animación, etc. para poder hacer que crear videojuegos para WebGL con unas características aceptables o incluso muy buenas sea una realidad.

Este hecho esta presente en la red y en las tecnologías que tenemos hoy en día.

Se ha conseguido ya realizar aplicaciones buenas y con características interesantes, y salen nuevas día a día, ademas de nuevas funcionalidades para crear contenido.

No obstante, como es lógico, dado que es una tecnología nueva y aun esta en desarrollo, estas tecnologías aun distan mucho de ser perfectas o muy completas, no hay mas que ver el hecho de que WebGL solo posee dos tipos de *shader* en la actualidad, vértice y fragmento, o el estado de ciertas librerías como Three.js, que no posee por ejemplo una funcionalidad para detección de colisiones, de modo que se tiene que recurrir a motores externos o bien crear el código que lo haga ad-hoc.

Sin embargo el futuro parece optimista, dado que se sigue una progresión siempre a mejorar, y cada poco tiempo salen nuevas funcionalidades que se implementan en las librerías existentes, dado que la comunidad se desvive por intentar mejorar la tecnología actual, tanto es así por parte de empresas como Google o Mozilla, que mejoran sus navegadores de Internet con intención de que en el futuro puedan proporcionar mejores experiencias 3D a sus usuarios y mas rápidas.

Aunque haya problemas con la tecnología actual, no es extraño esperar que sea normal, dado el estado de desarrollo experimental de muchas librerías y funcionalidades.

La aplicación que se ha desarrollado es un claro ejemplo de que se puede crear un videojuego 3D satisfactoriamente usando solo el propio WebGL y una librería como Three.js, con algunas librerías adicionales como Tween.js y software de modelado y animación 3D, además de que existe mucho material de modelos 3D y contenido multimedia gratuito en la red.

El estado que tendrá la tecnología en el futuro y lo que se podrá o se llegará a realizar no se conoce, pero es completamente lícito, dadas las circunstancias, ser optimista con ello.

5.2. Trabajo futuro.

En un futuro se puede, tomando como ejemplo la aplicación realizada en el presente trabajo, crear otras aplicaciones parecidas en 3D interactivo o incluso a partir de ella, ampliarla con mas niveles y enemigos, ya que es fácilmente ampliable.

Dado que existen otras librerías gráficas a parte de Three.js, de hecho existen numerosas librerías, sería interesante desarrollar nuevas aplicaciones con estas otras librerías para poder así ver el alcance de sus posibilidades actuales y comprobar que tipo de aplicaciones se pueden crear.

También es posible crear aplicaciones 3D en motores como Unity para luego portarlas a WebGL. Esto permite crear aplicaciones WebGL con herramientas consolidadas que simplifican muchas tareas de desarrollo, y esto podría ser también otro trabajo interesante.

En respecto a la aplicación desarrollada, hay muchas posibilidades de trabajo futuro, como por ejemplo crear mas niveles y enemigos como se ha comentado, añadir nuevas funcionalidades, menús, persistencia, optimizaciones, etc.

Otro trabajo posible seria comprobar mediante la realización de pruebas en varias máquinas diferentes el rendimiento de esta o cualquier otra aplicación 3D interactiva, dado que en el presente trabajo solo se han realizado experimentos en una única máquina, aunque ello dote de uniformidad a los casos de prueba, hacer experimentos como el realizado pero en diferentes máquinas y varias veces arrojaría mas información con la que extraer conclusiones.

Realizar un estudio teórico de la implementación de la librería o de otras librerías, y realizar diversos experimentos de diferente índole sobre ellas, daría a conocer el rendimiento o usabilidad de la propia librería o librerías en diferentes situaciones, lo cual sería también muy interesante de cara a nuevos desarrollos, en los que se debe seleccionar con qué librería trabajar.

También existe la posibilidad de tratar de introducir Realidad Virtual en la aplicación mediante el uso de WebVR, para así dotar a la aplicación de una nueva visión y paradigma con el cual el usuario puede interactuar con la aplicación.

Existe también la posibilidad de portar la aplicación a otros entornos como Unity o cualquier motor de videojuegos, para ver como se comporta en estos entornos de desarrollo y cual es su rendimiento en ellos.

Bibliografía

[1] threejs.org (2016) “Three.js official webpage with documentation and examples”, http://threejs.org/, Junio 2016.

[2] WebVR (2016) “Bringing Virtual Reality to the web”, https://webvr.info/, Agosto 2016.

[3] Rob Hawkes (2016) “Day 1: the future of games on the web”, http://12devsofxmas.co.uk/2012/12/day-1-the-future-of-games-on-the-web/, Agosto 2016.

[4] html5gamedevelopment (2016) “HTML5 Game Development”, http://html5gamedevelopment.com/, Agosto 2016.

[5] createjs (2016) “TWEENJS”, http://www.createjs.com/tweenjs, Agosto 2016.

[6] TurboSquid (2016) “3D models for professionals”, http://www.turbosquid.com/, Agosto 2016.

[7] Clara.io (2016) “3D models”, https://clara.io/scenes, Julio 2016.

[8] cgtrader (2016) “3D models”, https://www.cgtrader.com/, Agosto 2016.

[9] Freesound (2016) “Collaborative database of Creative Commons Licensed sounds”, https://www.freesound.org/, Agosto 2016.

[10] silicon (2016) “Página web con información sobre IT”, http://www.silicon.es/un-fallo-de-seguridad-en-webgl-afecta-a-chrome-y-firefox-55051, Septiembre 2016.

[11] GitHub (2016) “Fast, flexible, and collaborative development process”, https://github.com/, Junio 2016.

[12] Wikipedia (2016) “Wikipedia, la enciclopedia libre”, https://es.wikipedia.org/wiki/Wikipedia:Portada, Mayo 2016.

[13] Kuryanovich, Egor, “ Desarrollo de juegos en HTML5”. Madrid. Ediciones ANAYA multimedia, 2012.

[14] JavaScripter (2016) “Webpage to help JavaScript programmers develop better cross-platform applications”, http://www.javascripter.net/, Mayo 2016.