LIBRO 01

DISEÑO DE JUEGOS 3D PARA WEB

THREE.JS - HTML5 Y WebGL

www.TheFivePlanets.org

Jordi Josa

DISEÑO DE JUEGOS 3D PARA WEB – LIBRO 01 (PREVIEW)

THREE.JS – HTML5 Y WebGL

PREVIEW

Jordi Josa

ÍNDICE DE CONTENIDO

INTRODUCCIÓN

PARA QUIEN ES ESTE LIBRO

Este libro es para gente con conocimiento básico de JavaScript y HTML que quiera aprender cómo realizar escenas y animaciones en *3D* usando la librería *Three.js*, e iniciarse en el mundo del diseño de juegos sobre web. El libro forma parte de una colección en la que se irán introduciendo todos los conceptos, herramientas y tecnologías que necesitaremos para la creación de juegos complejos y convertirnos en verdaderos maestros.

Durante la colección de libros, diseñaremos e implementaremos un juego de rol "***THE FIVE PLANETS***". Por todo ello, el libro también va dirigido a aquellos que ya conocen *Three.js*, pero quieren seguir el proceso de creación de este juego en particular, y adquirir los elementos más complejos que se presentan en fases posteriores.

OBJETIVO DE LA COLECCIÓN

El objetivo de la colección de libros es aprender todo lo necesario para crear juegos en 3D con tecnología web y finalmente proponer formas de rentabilizar las creaciones. Para ello iremos creando desde el principio un juego de ROL, tipo "*The Elder Scrolls V: Skyrim*", "*Fallout 4*" ..., con los principales elementos que presentan, o sea la creación de los mapas en 3D, la inteligencia artificial de los enemigos, la gestión del inventario (armamento, armaduras, pociones, …), el sistema de misiones y por supuesto el sistema de estadísticas y niveles de los personajes tan característicos de estos juegos.

Bueno, quizás no tan ambicioso como los que he citado, pues unos juegos así requieren de equipos y presupuestos muy extensos con los que no contamos, sino más bien una mezcla de los juegos como eran antes, tipo "*Ishar III*", o la legendaria saga de los "*Migth and Magic*", con los de nueva generación.

CREEMOS NUESTRO JUEGO DE ROL

¿Por qué un juego de rol?

La creación de un juego de ROL en 3D presenta una colección de retos, que no se encuentran en los Juegos de plataforma, los árcades, o bien los de tipo tablero. Todos ellos mucho más sencillos, pues sólo cubren algunas funcionalidades.

Los juegos de ROL, o al menos en los que estamos pensado, incorporan todos estos elementos:

- Amplios mapas, con ciudades, bosques, llanuras y por supuesto mazmorras donde interactuar. Con ello cubriremos ampliamente el diseño de mapas y escenas en 3D, el examen de técnicas básicas para la creación del cielo y el suelo.
- Incluyen un "*shooter*" a tiempo real, con la posibilidad de lanzar hechizos devastadores, como bolas de fuego. En ellos nos enfrentamos a una horda de enemigos capaz de responder con ataques igual de formidables. Esto nos permitirá introducir la gestión de la Inteligencia Artificial, y las animaciones avanzadas.
- Las partidas suelen ser largas y demorarse por varios días. Con muchas misiones y un mundo muy cambiante. Todo ello implica que guardar una partida para posteriormente cargar los datos y volver al estado en que dejamos el juego implica el manejo de muchos datos. Ello nos brinda la posibilidad de presentar distintas técnicas para el almacenaje de datos en web.
- Estos juegos suelen disponer de una trama elaborada y, durante el juego, incluyen escenas cinematográficas y presentaciones para reforzar la historia. Algunos incluso incorporan escenas subidas de tono; ¿quién no recuerda el mítico "*Dragon Age*" con su sistema de romances entre personajes? Esto nos permitirá introducir el uso intensivo de cámaras.
- Otra característica es que cuentan con una *interfaz* y un sistema de menús elaborado. El diario de misiones, las fichas de los personajes, las pantallas de gestión del inventario, el mapa, entre otras. Ello nos permitirá mostrar distintas técnicas de cómo integrar la UI y hacerla interactuar con el mundo 3D.
- Los personajes cuentan con múltiples atributos y habilidades que gestionar. Conforme van avanzando de nivel ganan en fuerza, capacidad mágica y habilidades que requieren enemigos y retos más complejos. Este punto nos permitirá explicar cómo nivelar el juego para mantenerlo atractivo.
- Y si todo esto parece poco, suele haber un sinfín de elementos con los que interactuar como pasadizos secretos, palancas, cofres, plantas, libros, etc.

Por todo ello creemos que este tipo de juegos es una elección muy acertada como excusa para poder formarse.

¿Cómo es el juego que crearemos?

Al principio del libro el juego está bastante avanzado, tenemos ya montada buena parte de la *interfaz*, el sistema de colisiones, y un primer mapa de la ciudad donde nuestros héroes inician la aventura.

Hacer un proyecto así evidentemente va más allá de la capacidad de una sola persona, así que os confesaré algo, en realidad no tengo intención de finalizarlo, me conformo en crear un primer nivel y proponer a mis lectores que sean ellos quien finalmente lo lleven a su conclusión.

Os adjunto algunas de las capturas de pantalla de los elementos creados, con la intención de crear el gusanillo y despertar el interés en seguir la colección de libros.

Fig. I-01 Ejemplo del personaje Kiriela y del inventario.

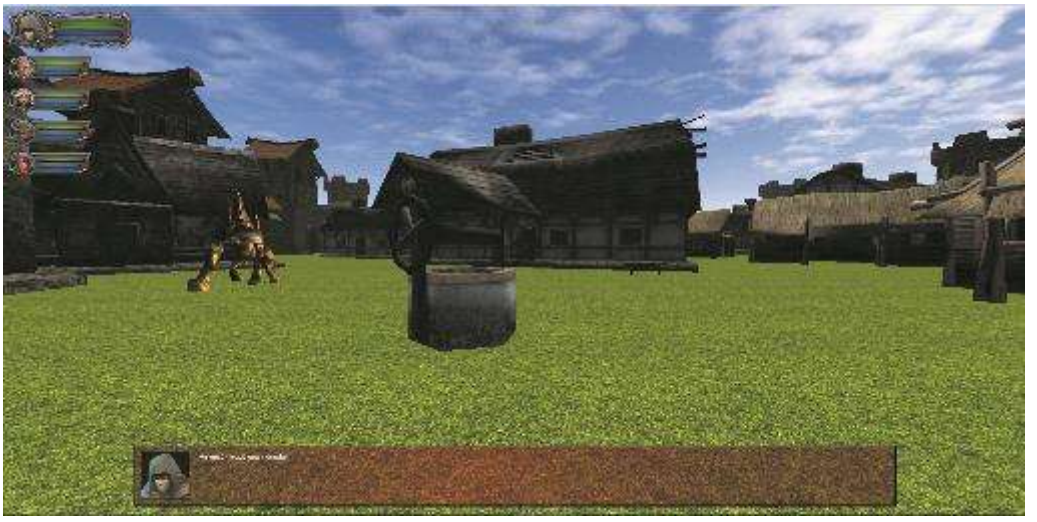

Fig. I-03 Imagen de la ciudad inicial.

Fig. I-4 Ejemplo de personajes animados en el mundo con los esqueletos en Blender.

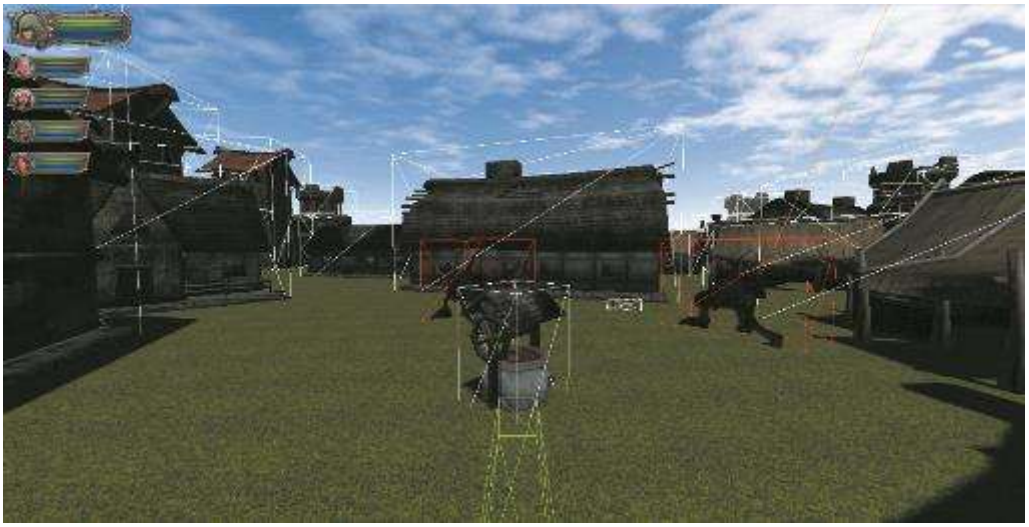

Fig. I-5 Ejemplo del sistema de colisiones.

Fig. I-5 Ejemplo de la interfaz de entrada.

TECNOLOGÍAS A USAR

Three.js

Es una biblioteca escrita en JavaScript para crear y mostrar gráficos animados en *3D* en un navegador *Web*. Puedes ver ejemplos y descargarla en http://threejs.org.

Esta biblioteca fue creada y liberada en *GitHub* por el español Ricardo Cabello en abril de 2010, conocido por su seudónimo de *Mr. Doob*. Con el tiempo se ha popularizado como una de las más importantes para la creación de las animaciones en *WebGL*. A día de hoy hay más de 390 colaboradores que se encargan de mejorarla.

Detección de colisiones y leyes de la física

Al desarrollar juegos o en ciertas escenas es necesario simular cómo las leyes de la física afectan a los objetos presentes. Para ello hay que tener en cuenta parámetros como la gravedad, la aceleración, el volumen, la velocidad, el peso o si las superficies son resbaladizas o rugosas. También hay que determinar qué pasa cuando dos objetos chocan entre sí, por ejemplo, las bolas de fuego que lanzan los magos al impactar contra los enemigos. Por suerte disponemos de varias librerías JavaScript que realizan buena parte de los cálculos para nosotros.

Ammo.js

Ammo.js se basa en la librería *BulletPhysics* (http://bulletphysics.org/) escrita en C++. El código fuente se ha traducido directamente a JavaScript sin intervención humana, por lo que la funcionalidad es idéntica a la original, pero su rendimiento se ve afectado por las diferencias entre los lenguajes de programación.

Cannon.js

Cannon.js es una librería creada directamente en JavaScript inspirándose en *ammo.js*. Su rendimiento es ligeramente mejor al estar escrita directamente con JavaScript. Puedes ver ejemplos y descargarla en http://cannonjs.org.

Physijs

Physijs es una capa superior para la librería *ammo.js* (aunque también existe una rama *cannon.js*). Permite la simulación física en un hilo separado (vía "*webWorker*") y así evita el impacto en el rendimiento de los cálculos en la aplicación y permite acelerar el tiempo de renderizar el 3D. Puedes ver ejemplos y descargarla en http://chandlerprall.github.io/Physijs/

Oimo.js

Oimo.js al igual que las dos librerías anteriores es un motor para la simulación física de objetos rígidos. Es una conversión completa de JavaScript de *OimoPhysics* creada originalmente para *ActionScript* 3.0. Puedes ver ejemplos y descargarla en https://github.com/lo-th/Oimo.js/

JQuery

JQuery es una biblioteca de JavaScript, que facilita la interacción con los documentos HTML. Dispone de métodos para manipular el árbol DOM, manejar eventos, desarrollar animaciones e implementar las llamadas AJAX en páginas web.

En algunos de nuestros ejemplos la usaremos para implementar los menús y pantallas secundarias del juego, como el inventario, el diario de misiones, o las ventanas flotantes. Puedes ver ejemplos y descargarla en https://jquery.com/

Blender

Blender es un programa informático multiplataforma, dedicado especialmente al modelado, iluminación, renderizado, animación y creación de gráficos tridimensionales. Durante la colección de libros, lo usaremos para modificar y crear objetos en *3D*, y también para animar los monstruos y personajes del juego.

La aplicación es distribuida de forma gratuita. Actualmente es compatible con *Windows*, *Mac*, *Linux* (Incluyendo *Android*), *Solaris*, *FreeBSD e IRIX*. Puedes ver ejemplos y descargarla en https://www.blender.org/

Adobe Photoshop

Durante el proceso de creación de un juego será necesario estar acompañados por un buen editor de gráficos. Nos servirá para crear y editar las texturas de los objetos y los personajes, y para crear la interfaz gráfica del juego.

Corregir las gamas de colores, reemplazar un color por otro o ajustar los tamaños de las imágenes para que sean lo suficientemente pequeñas para la web, serán tareas básicas en la preparación de los objetos. Por todo ello es imprescindible contar con un buen editor que permita todo esto cómodamente. Personalmente, para este libro he usado Photoshop, ya que es uno de los más completos y funcionales del mercado, lamentablemente es la única herramienta que usaremos que no es gratuita.

C1 - THREE.JS: PRIMEROS PASOS

En este capítulo presentaremos los elementos más básicos de *Three.js* y crearemos nuestra primera escena.

HOLA MUNDO

Los navegadores modernos incorporan funcionalidades para la composición de imágenes en *2D* y *3D*.

Three.js es una librería JavaScript que facilita el desarrollo de escenas en *3D*, incorporando una capa para superar el problema de las pequeñas diferencias de implementación que hay entre los distintos navegadores. Para este libro hemos utilizado la versión 80, que puedes descargar en la página http://threejs.org.

Como primer paso vamos a crear una escena que mostrará un cubo rotando y moviéndose horizontalmente. Este ejemplo nos servirá para introducir los elementos más básicos que incorpora la librería. Recuerda que puedes descargar los códigos de ejemplo de este libro en https://www.thefiveplanets.org/blog/book01.

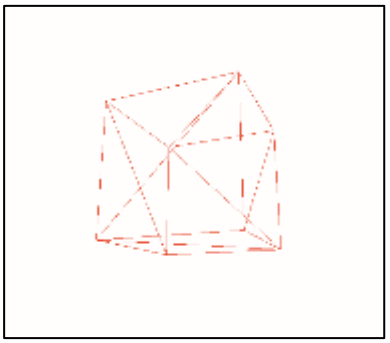

Fig. 1.1 Cubo en rotación.

El código completo para crear nuestra escena es el siguiente.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
 <style>
  body {
    background-color: #ffffff;
    margin: 0;
    overflow: hidden;
```

```
  }
 </style>
</head>
<body>
<script src="https://ajax.googleapis.com/ajax/libs/threejs/r76/three.min.js "></script>
<script>
   var camera, scene, renderer;
   var geometry, material, mesh;
   var clock;

      function init() {
         renderer = new THREE.WebGLRenderer();
         renderer.setSize( window.innerWidth, window.innerHeight );
         document.body.appendChild( renderer.domElement );

         scene = new THREE.Scene();

         geometry = new THREE.CubeGeometry( 1, 1, 1 );
         material = new THREE.MeshBasicMaterial(
              {color: 0xff0000, wireframe: true});

         mesh = new THREE.Mesh( geometry, material );
         scene.add( mesh );

         camera = new THREE.PerspectiveCamera(
               75,window.innerWidth/window.innerHeight,0.1,100 );
         camera.position.set(0,0,-3);
         camera.lookAt(mesh.position);

         clock = new THREE.Clock();
         window.addEventListener( 'resize', onWindowResize, false );
      }
      var dir=1;
      function animate() {
         requestAnimationFrame( animate );

         var delta =  clock.getDelta();

         mesh.rotation.x += delta * 0.5;
         mesh.rotation.y += delta * 2;
         mesh.position.x += dir*delta;
         if (mesh.position.x > 2) {
            dir=-1;
         } else if (mesh.position.x < - 2)  {
            dir=1;
```

```
        }
        renderer.render( scene, camera );
    }

    function onWindowResize() {
        windowHalfX = window.innerWidth / 2;
        windowHalfY = window.innerHeight / 2;
        camera.aspect = window.innerWidth / window.innerHeight;
        camera.updateProjectionMatrix();
        renderer.setSize( window.innerWidth, window.innerHeight );
    }

    init();
    animate();

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

Para ver el código del ejemplo en funcionamiento y descargarlo puedes usar el siguiente enlace:

http://www.thefiveplanets.org/b01/c01/01-hello-threejs.html.

Al ejecutarlo verás un cuadrado con los bordes rojos rotando sobre sí mismo y moviéndose de derecha a izquierda y viceversa. A continuación, vamos a dar una ojeada a los distintos elementos que componen el ejemplo.

ESCENA

La escena es la composición del mundo que queremos mostrar, en ella iremos añadiendo todos los elementos que lo conforman, es decir:

- los objetos en 3D
- la cámara o cámaras por donde ver el mundo
- los puntos de luz o luces que iluminan la escena
- los sonidos y la música ambiente
- y configuraremos los efectos especiales como es la niebla, …

La instrucción para crearla es:

```
scene = new THREE.Scene();
```

MESH

A los objetos en 3D que añadimos a la escena les llamaremos *Mesh*. En nuestro caso es el cubo.

Todo objeto está compuesto como mínimo por la geometría (su forma) y el material que indica el color, las texturas del mismo, o incluso cómo le afecta la iluminación.

```
geometry = new THREE.CubeGeometry( 1, 1, 1 );
material = new THREE.MeshBasicMaterial({
    color: 0xff0000,
    wireframe: true});
mesh = new THREE.Mesh( geometry, material );
scene.add( mesh );
```

El *Mesh* también puede componerse de la información sobre cómo debe animarse el objeto. Por ejemplo, la figura de un campesino podría contener la secuencia de animaciones para andar, correr, labrar, pelear… que indican cómo debe ir deformándose el objeto para crear la sensación de movimiento.

Geometría

Las geometrías son instancias de ***THREE.Geometry*** y están formadas por vértices y caras. Los vértices son instancias de ***THREE.Vector3*** y representan puntos en un espacio tridimensional, mientras que las caras son triángulos que unen tres puntos y son instancias de ***THREE.Face3***. Así, por ejemplo, una esfera en realidad es una colección de triángulos unidos entre sí. En el caso del cubo tendremos 8 vértices, y 12 triángulos (caras), cada lado del cubo está compuesto por dos triángulos.

En los siguientes enlaces vemos el cubo con las caras y los vértices resaltados. En el primero los vértices están resaltados con puntos rojos, en el segundo las caras están pintadas de distintos colores.

Vertices: https://www.thefiveplanets.org/b01/c01/02-vertices.html

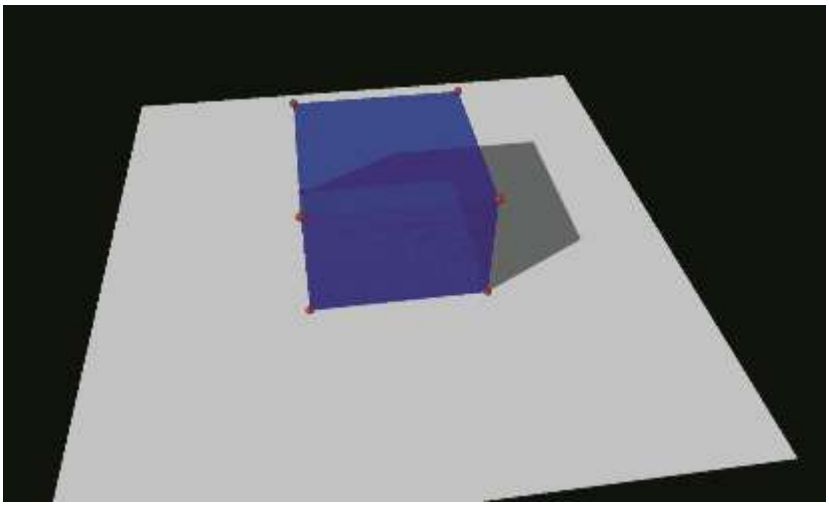

Fig. 1.2 Muestra los vértices de un cubo.

Caras: https://www.thefiveplanets.org/b01/c01/03-faces.html

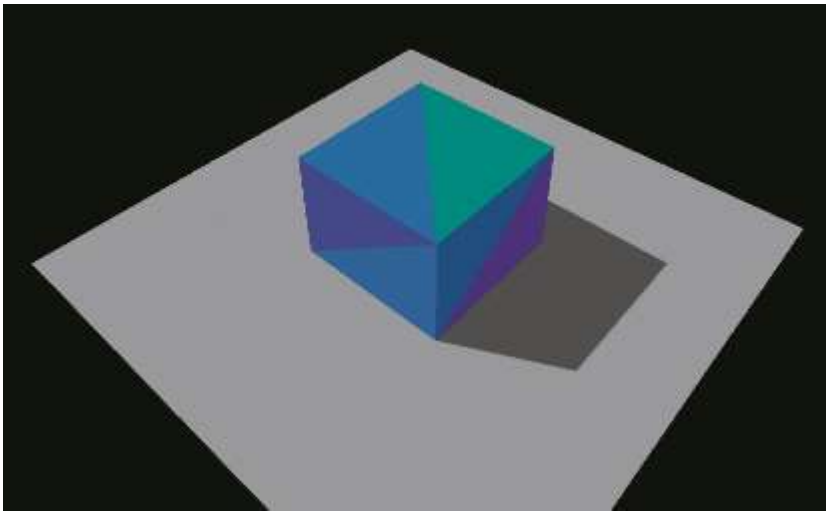

Fig. 1.3 Muestra las caras de un cubo.

En el siguiente ejemplo vemos la representación de un triángulo plano.

```
var geometry = new THREE.Geometry();
geometry.vertices.push(
    new THREE.Vector3( -1,  1, 0 ),
    new THREE.Vector3( -1, -1, 0 ),
    new THREE.Vector3(  1, -1, 0 )
);
geometry.faces.push(
    new THREE.Face3( 0, 1, 2 )
);
var material = new THREE.MeshNormalMaterial();
var mesh = new THREE.Mesh( geometry, material );
scene.add( mesh );
```

Usa el siguiente enlace para ver el código en funcionamiento: https://www.thefiveplanets.org/b01/c01/04-geometry.html.

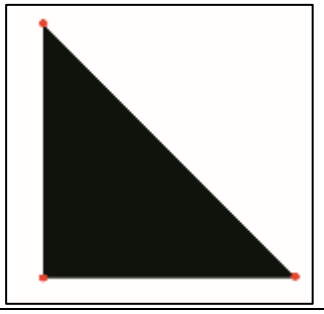

Fig. 1.2 Triangulo con los vértices representados con un círculo rojo.

Normalmente, rara vez definiremos las geometrías de forma manual. Por suerte *three.js* incluye muchas formas por defecto, como cubos, esferas, cilindros, planos, … En el capítulo dos puedes ver una lista completa.

Blender es un editor gratuito que permite crear figuras avanzadas y exportarlas en formatos que nos permiten usarlas directamente en *three.js*.

Material

Los materiales son "*la piel*" de las figuras; sirven para definir de qué color es cada cara de una figura, cómo la luz actúa en dicha cara, o si son visibles ambas caras o solo una. Al igual que con las geometrías, *Three.js* nos proporciona una rica colección de clases para crear distintos tipos de materiales según el efecto que queramos generar, como por ejemplo materiales que ignoran la luz.

En el ejemplo inicial hemos usado el material más básico ***THREE.MeshBasicMaterial***, indicando el color rojo y la propiedad *wireframe* a *true* para que en vez de pintar las caras de un color solido se muestren las líneas entre los vértices.

```
material = new THREE.MeshBasicMaterial({
    color: 0xff0000, wireframe: true}
);
```

Algunas propiedades con las que puedes ir experimentando son *transparent* con valor *true* y *opacity* para indicar el grado de transparencia de las capas.

Existe una propiedad que es importante que conozcas: *side*, que permite indicar qué cara se muestra. Normalmente, por motivos de rendimiento, sólo se suelen pintar las caras del objeto que sabemos que se van a visualizar. Por ejemplo, en un cubo cerrado, si no vamos a enfocar nunca el interior sólo mostraremos las caras externas.

Los tres valores que permite son ***THREE.FrontSide***, por delante; ***THREE.BackSide***, por detrás; y ***THREE.DoubleSide***, por ambos lados. Que se muestre la cara delantera o la trasera no depende de la posición de la cámara. Dichas caras dependen del orden de cómo hayas dibujado sus vértices. La cara delantera es la que corresponde con la dirección inversa de las agujas del reloj. Si usamos *THREE.DoubleSide*, siempre será visible, indistintamente de donde coloquemos la cámara.

CÁMARA

Las cámaras son los ojos con los que podemos ver el mundo, no tienen una representación gráfica. Una escena puede contener tantas cámaras como deseemos, pero sólo se podrá haber una de ellas activa para visualizar el mundo, pudiendo alternar con otra sin limitaciones.

Puedes girarlas y posicionarlas, pero los resultados de dichos cambios sólo se verán al llamar el método *render*, tal y como hemos visto en los anteriores ejemplos.

Existen dos tipos de proyecciones que podemos usar en el momento de crear la cámara:

La proyección en perspectiva (***THREE.PrespectiveCamera***) deforma los objetos según la distancia y posición que se encuentren con respecto a la cámara, tal y como ocurre en el mundo real. Es la que se suele usar en juegos en primera persona.

La proyección Isométrica (***THREE.OrthographicCamera***) es una perspectiva que respeta el tamaño de los objetos, independientemente de la distancia a la que se encuentren de la cámara (2.5D). Se suele usar en juegos como "*Diablo*" o en muchos juegos de rol diseñados en HTML.

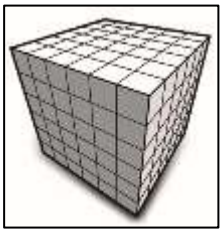

Fig. 1.5 Visión del mudo con proyección en Perspectiva.

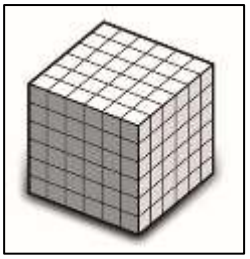

Fig. 1.6 Visión del mudo con proyección Isométrica.

Cámara en perspectiva (THREE.PrespectiveCamera)

Los cuatro parámetros de la cámara en perspectiva usada en el ejemplo inicial son los siguientes:

```
camera = new THREE.PerspectiveCamera(
    75, window.innerWidth/window.innerHeight, 0.1, 100);
```

Teniendo estos cuatro parámetros en mente observemos esta figura.

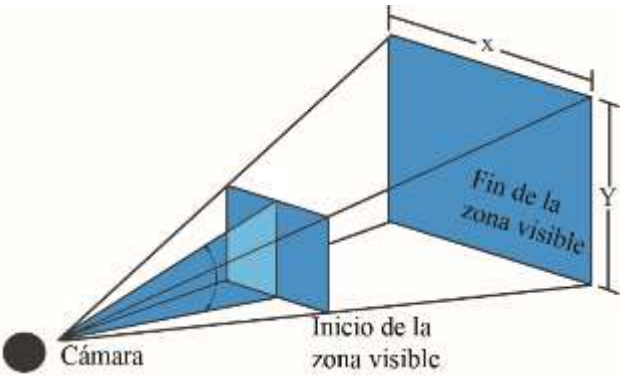

Fig. 1.7 Parámetros cámara.

- El primer parámetro (***75***) define el campo visual vertical de la cámara en grados (desde abajo hacia arriba). Se trata de la extensión del mundo observable que se ve en la pantalla en un momento determinado. El campo visual horizontal se calcula con el vertical.
- El segundo parámetro (***window.innerWidth / window.innerHeight***) define la relación de aspecto de la cámara. Por lo general se usa el ancho de la ventana dividido por su altura. De otro modo la imagen se ve deformada.
- El tercer parámetro (***0,1***) define el inicio de la zona visible. Es decir, desde la cámara hasta la distancia indicada los objetos no son visibles. En el ejemplo la distancia es prácticamente cero.
- El último parámetro (***100***) define el fin de la zona visible. En el ejemplo, cuando un objeto sobrepasa las +100 unidades estará fuera de la zona visible de la cámara y sólo se mostrará la parte que esté dentro del área.

Otras propiedades importantes de la cámara son su posición y el punto de la escena al que enfocar.

Todos los objetos (cámaras, figuras, luces...) y escenas que creemos tienen una propiedad llamada ***position*** que contiene una instancia de un vector (***THREE.Vector3***). Para indicar la posición de la cámara, basta con llamar el método ***set*** de ***THREE.Vector3*** pasando como parámetros las coordenadas (*X,Y,Z*).

```
camera.position.set (0,0,-3);
```

Para girar la cámara podemos usar el método ***lookAt*** para que enfoque a un punto concreto. Este método sólo funcionará si la cámara ha sido añadida directamente a la escena y no a un objetivo.

```
camera.lookAt(mesh.position);
```

Para ver en funcionamiento los dos tipos de cámara puedes usar el enlace que adjunto a continuación. En el ejemplo podemos alternar de una perspectiva a otra.

https://www.thefiveplanets.org/b01/c01/05-camera.html.

RENDER

Llegados a este punto ya tenemos todo preparado para mostrar una bonita escena en 3D. Únicamente nos falta indicar la parte de la página web donde dibujar nuestra composición. Para ellos usaremos los "***renders***", que se encargarán de crear elemento DOM (*WebGL, Canvas o CSS3*) y añadirlo en la página.

```
renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
renderer.setClearColor(new THREE.Color(0xEEEEEE, 1.0));
document.body.appendChild( renderer.domElement );
```

Para crear un *render* necesitamos al menos indicar las siguientes propiedades:

1. El tamaño en píxeles que tendrá. En el ejemplo hemos usado la anchura y la atura de la ventana (*window.innerWidth, window.innerHeight*).
2. El color de fondo que utiliza Three.js. Al indicar el color podemos también indicar la opacidad, en nuestro caso hemos indicado un color solido (1.0)

El *render* crea un elemento de *HTML5* de tipo *CANVAS* que debemos de añadir a nuestra página web. Podemos añadirlo directamente al cuerpo del documento (***document.body.appendChild***), o bien crear una capa (***div***) con un "*id*" para poderla referenciar.

```
document.getElementById("id").appendChild(render.domElement);
```

Cada vez que queramos que los cambios realizados en la escena se reflejen por pantalla debemos llamar el método *render*:

```
render.render(escena, camara);
```

Como parámetros indicamos: la escena que queremos renderizar y la cámara. Evidentemente una misma escena se verá diferente según la posición que ocupe la cámara y la dirección en la que esté enfocada.

Pista

Usando varios *renders*, uno encima del otro marcando el color de fondo como transparente, podemos mejorar el rendimiento. Por ejemplo, en el superior se incluirían las animaciones de los personajes, mientras que en el inferior representaríamos el fondo.

WebGL (THREE.WebGLRenderer)

Este es el tipo de *render* que hemos usado en el ejemplo inicial, y lo vamos a utilizar de forma habitual. Para representar las escenas utiliza las *APIs* de *WebGL* presentes en los navegadores modernos. De todos los disponibles es el único que permite los efectos avanzados de sombras e iluminaciones.

Canvas (THREE.CanvasRenderer)

El *CanvasRender* utiliza las *APIs* de Canvas 2D sin utilizar WebGL. Esto implica que para escenas sencillas se puede llegar a una gama más amplia de dispositivos y navegadores, pero resulta mucho más lento ya que no se beneficia de la aceleración por hardware que ofrecen las *APIs de WebGL*.

En nuestro caso no lo usaremos, ya que para el tipo de juego que queremos crear es necesario recurrir a todas las técnicas de optimización.

Puedes ver un ejemplo en el siguiente enlace:
https://www.thefiveplanets.org/b01/c01/06-canvasrenderer.html

CSS3D (THREE.CSS3DRenderer)

La conjunción de HTML y CSS cada vez dispone de más posibilidades. Las últimas versiones de CSS soportan transformaciones 3D, por lo que podemos aplicar los estilos CSS para transformar nuestras páginas WEB en objetos 3D.

Fig. 1.8 Ejemplo del CSS3DRender

A diferencia de los otros dos *renders* no utiliza el elemento HTML CANVAS, sino que directamente son los elementos HTML a los que aplica transformaciones y estilos CSS. Obviamente, las posibilidades no son las mismas que los otros dos.

Puedes ver el resultado de aplicar esta tecnología a una simple página web con un formulario: http://www.thefiveplanets.org/b01/c01/07-css3drenderer.html

El *render* no está incluido en la carpeta de la librería, sino que directamente se adjunta como un ejemplo por lo que puedes descargarlo en la siguiente URL:

https://raw.githubusercontent.com/mrdoob/three.js/master/examples/js/renderers/CSS3DRenderer.js

EJES, POSICIÓN, ESCALA Y ROTACIÓN

Ejes

Three.js usa un sistema diestro de coordenadas. La pantalla del dispositivo coincide con el plano *xy* y los puntos del eje *z* positivo apuntan fuera de la pantalla, hacia los ojos del observador.

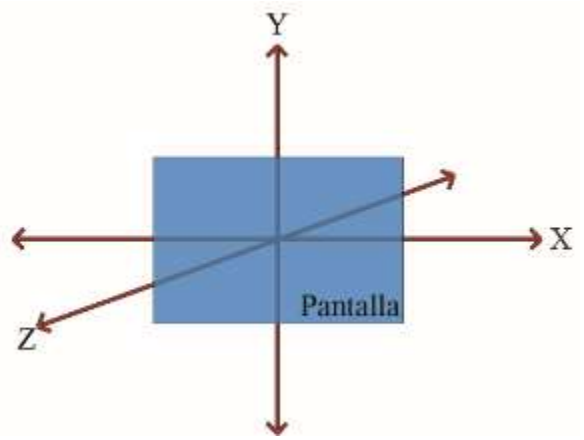

Fig. 1.9 Coordenadas

Cuando se agrega un objeto de *Three.js* dentro de una escena, este se posiciona (de forma predeterminada) en el origen del sistema de coordinadas *xyz (0,0,0)*. Por lo tanto, si añades una cámara y un cubo a una escena, ambos se situarán en la posición (*0, 0, 0*) y veremos el cubo desde dentro. En este caso, la solución es mover la cámara o el cubo, por ejemplo:

```
camera.position.z = 50
```

Posición y Escala

Una de las múltiples posibilidades para modificar la posición y el tamaño del objeto son las propiedades ***position*** y ***scale***. Ambas son instancias de *THREE.Vector3* que

se crean indicando un punto *(x,y,z)*. Para la posición el valor por defecto es (*0,0,0*) que es en el origen del sistema de coordenadas, mientras que para la escala es (*1,1,1*) que indica que el objeto debe mantener el tamaño original respecto a los tres ejes. Por ejemplo, con (*1,2,1*) estamos indicando que el objeto debe deformase para ser el doble de alto, manteniendo la misma anchura y profundidad. Si por el contrario indicamos (*2,2,2*) estamos indicando que duplicará el tamaño conservando las mismas proporciones.

```
var vec=new THREE.Vector3(x,y,z)
```

La clase *THREE.Vector3* dispone de las siguientes propiedades que se referencian a continuación. Para ver una lista completa puedes consultar la documentación.

Propiedades
X: Valor de la coordenada X.
Y: Valor de la coordenada Y.
Z: Valor de la coordenada Z.
Métodos
.set (x, y, z): Establece los valores para los tres ejes.
.setX (x): Establece el valor para el eje de las X.
.setY (y): Establece el valor para el eje de las Y.
.setZ (z): Establece el valor para el eje de las Y.
.copy (v): Dado otro vector copia los valores de los tres ejes.

En el ejemplo inicial accedemos directamente a las propiedades para establecer los valores, pues es la forma más sencilla.

```
mesh.position.x += dir*delta;
```

Posición relativa

Todos los objetos disponibles para añadir a la escena (la cámara, las figuras, las luces, e incluso el audio) son instancias de ***THREE.Object3D***, los cuales tienen el método ***Add*** que permite añadir otros objetos dentro del objeto padre. En este caso la posición del hijo será relativa no respecto al eje de coordenadas global, sino sobre la posición del padre. Lo mismo sucede con la escala y la rotación.

https://www.thefiveplanets.org/b01/c01/08-hello-threejs-scale.html

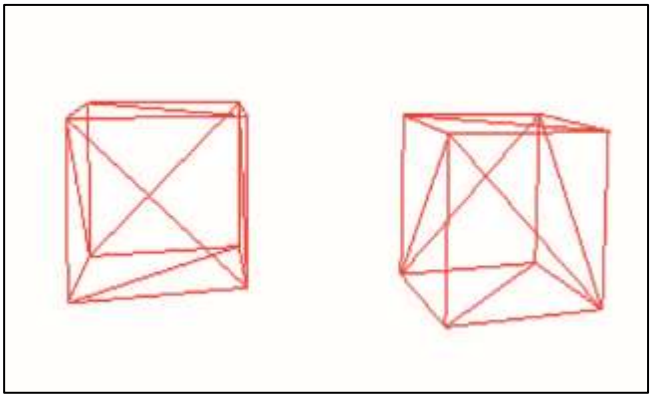

Fig. 1.10 Cubo añadido dentro de otro cubo

Pista

Recuerda que puedes añadir objetos dentro de otros objetos. En el caso que cambies la posición, rotación o escala del objeto padre, los hijos también serán afectados.

Rotación

La propiedad "***rotation***" nos permite modificar la rotación de un objeto. Su valor es instancia de la clase ***THREE.Euler***, que se define indicando los siguientes parámetros (*x, y, z, orden*). Donde "*x*" indica los radianes a girar con respecto a su eje *x*; "*y*" y "*z*" son iguales, pero respecto a sus correspondientes ejes; el *orden* versa sobre el orden de giro, por ejemplo '*XYZ*' indica que primero se rotará respecto al eje *X*, después al *Y* e finalmente al *Z*.

```
var eu=new THREE.Euler( 0, 1, 1.57, 'XYZ' );
```

La clase THREE.Vector*3* dispone de las siguientes propiedades, para ver una lista completa puedes consultar a la documentación:

Propiedades

x: Valor de la coordenada X.
y: Valor de la coordenada Y.
z: Valor de la coordenada Z.
order: Orden de los ejes, por defecto "*XYZ*" en mayúsculas.

Métodos

.set (x, y, z, order): Establece los valores para los tres ejes.
.copy (eu **):** Dado otro *euler* copia los valores de los tres ejes y su orden.

Rotación sobre su eje

La forma más fácil para rotar sobre un eje es como lo hemos realizado en el ejemplo inicial, accediendo a las propiedades *x, y, z* de *rotation*.

```
mesh.rotation.x += delta * 0.5;
mesh.rotation.y += delta * 2;
```

En caso de rotar sobre varios ejes debemos vigilar el orden, no es lo mismo girar 90º sobre el *X* y después 90º sobre el *Y*, que hacerlo al revés. Fíjate que el orden de las instrucciones no variará el resultado, ya que es el método *render* que aplicará el giro en base la propiedad *order*.

Rotación respecto a un punto de referencia

Como hemos visto los objetos se pueden añadir dentro de otros creando una jerarquía de padres e hijos. De esta forma la posición del hijo es relativa al punto central del padre. Si giramos el padre, el hijo girará, no respecto a su eje sino el del padre. Así pues, para girar un objeto respecto a un punto, basta con crear una instancia de *THREE.Object3D* (sin geometría, ni materiales), añadir el hijo a una posición distinta de (*0,0,0*), y girar el padre sobre su hijo.

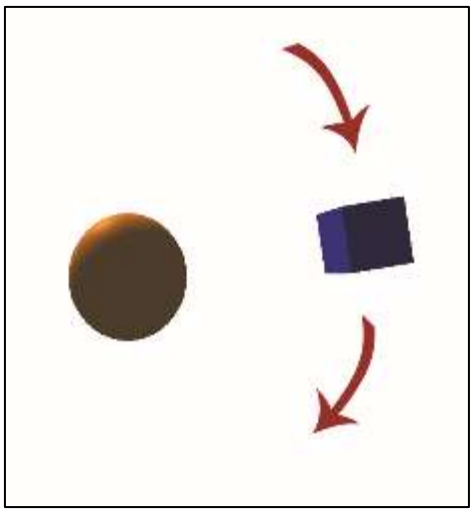

Fig. 1.10 Ejemplo de rotación respecto a un punto de referencia

En este enlace puedes ver el ejemplo:

https://www.thefiveplanets.org/b01/c01/09-rotate-object-around-point.html

Los pasos que hemos seguido son:

1. Creamos el punto respecto al que gira el objeto y lo situamos en el espacio.

```
pivotPoint = new THREE.Object3D();
pivotPoint.position.set(0,0,0);
scene.add( pivotPoint );
```

2. Creamos el objeto a girar.

```
geometry = new THREE.CubeGeometry( 0.5, 0.5, 0.5 );
material = new THREE.MeshPhongMaterial(
    { color: 0x0000ff}
);
mesh = new THREE.Mesh( geometry, material );
```

3. Añadimos el objeto al "*pivotPoint*".

```
pivotPoint.add(mesh);
mesh.position.set(0,2,0);
```

4. Giramos al padre.

```
pivotPoint.rotation.z += delta * 2;
```

Convertir grados a radianes y viceversa

Las funciones para la rotación en *Three.js* utilizan radianes, pero es más habitual que pensemos en grados. Para convertir grados a radianes sólo es necesario aplicar la siguiente formula:

```
radianes = grados*Math.PI/180;
```

Por suerte *Three.js* incorpora las siguientes funciones por defecto que realizan el cálculo por nosotros:

THREE.degToRad (grados): Convierte grados a radianes.
THREE.radToDeg (radianes): Convierte radianes a grados.

ANIMACIÓN DE LA ESCENA

Para mover las figuras durante la ejecución, no hay más que modificar su atributo *position* o *rotation*, y luego llamar al método *render* periódicamente. Una animación (como las películas y los juegos) no son más que sucesiones de imágenes que al reproducirlas en secuencia rápidamente dan sensación de movimiento. Cuanto más suave sea el cambio de posición y rotación de la figura en cada imagen, más real se verá la animación, por ello un factor crítico es el número de fotogramas que se muestran en cada segundo (*FPS*). El cine, por ejemplo, funciona a 24 fotogramas (aunque *El Hobbit* de Jackson usa 48 fotogramas, y quizás los demás le copien). Los juegos, sin embargo, deben garantizar un mínimo de 30 FPS, aunque lo normal es que suelan estar entre 50 y 60.

En JavaScript existe la función ***setTimeOut*** que permite ejecutar una función al cabo de un número fijo de milisegundos, con ello podemos realizar un código que actualice la escena y ejecute al método *render* y vuelva a llamar recursivamente el *setTimeOut*, de forma que se cree un bucle infinito de llamadas. Este mismo efecto se puede lograr con la función ***setInverval***, que permite indicar una función y un tiempo en milisegundo con la periodicidad en que debe repetirse.

Ambas funciones presentan problemas a la hora de animar, sobretodo ***setInterval*** que, al llamarse de forma constante, si el renderizado se demora mucho tiempo las llamadas pueden ir produciéndose una detrás de la otra sin dar tiempo a realizar el resto de tareas, provocando el colapso del sistema. Otro problema que sí afecta a ambas funciones es que siempre se ejecutarán independientemente de que el usuario esté visualizando otra página web, o de que la escena no esté visible por pantalla en ese momento, lo cual consume tiempo de *CPU* y *GPU* innecesariamente. Para móviles y ordenadores portátiles supone que consuman más batería y se calienten, mientras que en los de sobremesa provocará un mayor uso del ventilador. Finalmente está la potencia del dispositivo que, según la tarjeta gráfica y el tipo de dispositivo, soportará un grado mayor o menor de actualizaciones por segundo. Por lo que podemos estar o forzando demasiados intentos de renderizado o quedándonos por debajo del óptimo.

Por suerte se ha introducido una nueva función JavaScript llamada "***requestAnimationFrame***". Esta función es similar al *setTimeout* pero mejorada. La función sólo se ejecutará cuando nuestro *CANVAS* esté total o parcialmente visible y, por si fuera poco, gestionará por nosotros la frecuencia con la que se llama la función de renderizado.

```
function animate() {
   requestAnimationFrame( animate );
   ...
   renderer.render( scene, camera );
}
```

En el ejemplo, la función que llama a *requestAnimatioonFrame* es ***animate*** la cual es invocada por primera vez al cargar la página con el *onload*; a partir de aquí la función se llamará de forma repetitiva. Observa que en el código vamos incrementando o decrementando la posición y el ángulo de rotación, que es lo que provoca la sensación de movimiento.

Clock

Lamentablemente, con lo que acabamos de ver, aún no se han terminado los problemas. La periodicidad en que se ejecuta la función dependerá del dispositivo y de las tareas paralelas que se estén ejecutando. Habitualmente será 60 veces por segundo, pero ello no está garantizado, pudiendo bajar. Ello producirá que la

animación se ejecute muy lentamente o muy rápidamente, lo que para nuestros objetivos no es aceptable. Por ejemplo, la animación de andar o correr requiere que se ejecute con un tiempo determinado para parecer realista.

Three.js incorpora la clase ***THREE.Clock*** con el método ***getDelta()*** que nos permite calcular los segundos que han pasado desde una ejecución a la otra. De esta forma, si queremos que un objeto rote 2 unidades cada segundo, multiplicaremos *2*delta*, donde delta es el resultado del método ***getDelta()***.

```
...
clock = new THREE.Clock();
...
function animate() {
        requestAnimationFrame( animate );

        var delta =  clock.getDelta();
        mesh.rotation.x += delta * 0.5;
        mesh.rotation.y += delta * 2;
        mesh.position.x += dir*delta;
        if (mesh.position.x > 2) {
           dir=-1;
        } else if (mesh.position.x < - 2)  {
           dir=1;
        }
        renderer.render( scene, camera );
}
```

C2 - TREEJS: PREPARANDO EL ENTORNO DE DESARROLLO

Cuando desarrollamos soluciones con *Three.js* la forma más simple para probar los resultados es abriendo el fichero local con el navegador. Desafortunadamente la solución no funcionará en todos los casos, generando un error de seguridad cuando cargamos texturas con imágenes y modelos.

Lo más aconsejable es trabajar con un servidor web instalado localmente, pues generará una experiencia lo más próxima a la que obtendrá el usuario final. A continuación, veremos varias propuestas para ejecutar nuestras creaciones sin problemas.

SOLUCIONAR EL ERROR "CROSS-ORIGIN-DOMAIN"

Este error de seguridad es fácilmente reproducible si abres el siguiente enlace https://www.thefiveplanets.org/b01/c02/01-solve-cross-origin-issues.html.

Al ejecutarlo producirá un error como el que se muestra en la figura *2.1*. El mensaje informativo contendrá referencias al error de orígenes cruzados (*cross-origen*) o error de seguridad (*security-error*) según el navegador que usemos.

Fig. 2.1 Error visto en Chrome

El navegador nos da este mensaje y detiene la ejecución del JavaScript para evitar accesos a los datos personales e impedir así ataques contra nuestro equipo. Cuando estamos desarrollando y sabemos que las fuentes de los archivos son seguras, nos puede convenir desactivar esta comprobación.

Con *Chrome* y *Firefox* podemos configurar fácilmente el navegador para permitir el acceso a los ficheros desde llamadas Ajax. Recuerda que si ejecutas un servidor web local estos pasos no son necesarios.

Chrome

Para abrir *Chrome* desactivando la comprobación debemos indicar el siguiente parámetro "***--allow-file-access-from-files***" al momento de ejecutarlo.

Según el sistema operativo que utilicemos el comando será distinto:

Para Windows llama el siguiente comando:

```
chrome.exe --allow-file-access-from-files
```

También puedes crear un fichero "***.bat***"; por ejemplo "runchrome.bat" con el siguiente contenido:

```
Start  "Chrome" "chrome.exe" --allow-file-access-from-files
Exit
```

Para Linux el comando es el siguiente:

```
google-chrome --allow-file-access-from-files
```

Para Mac la orden a escribir es:

```
open -a Google\chrome --args --allow-file-access-from-files
```

Recuerda que el parámetro sólo se aplicará si Chrome no está en ejecución en ese momento, por lo que previamente deberás cerrar cualquier instancia del programa. Una vez finalices las pruebas cierra el Navegador Chrome, ya que, en este modo, el sistema es más vulnerable a ataques.

Firefox

Para Firefox es necesario abrir el navegador y realizar los siguientes pasos:

1. Introduce la URL ***about:config***; a continuación, te aparecerá un mensaje de advertencia indicando que los cambios que realices pueden tener un impacto negativo en la seguridad del equipo y el rendimiento de la aplicación. Pulsa el botón para continuar.

2. Busca ***security.fileuri.strict_origin_policy*** y modifica su valor a false.

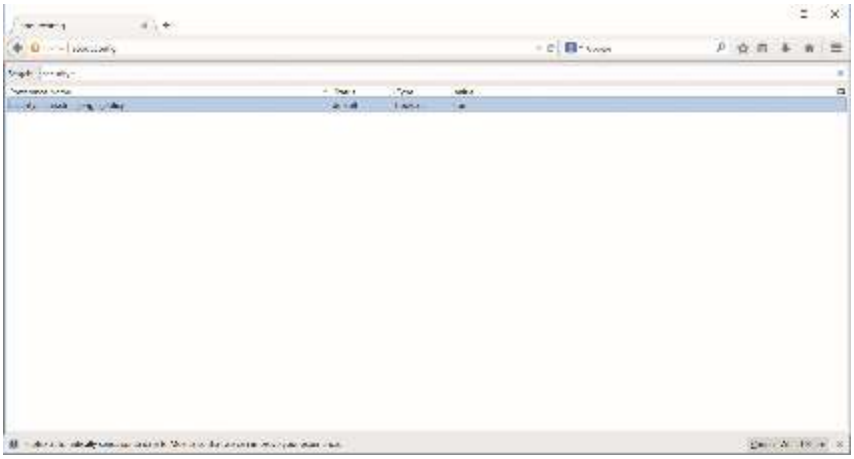

Fig. 2.2 Pantalla de configuración para Firefox

3. Ahora ya puedes abrir el enlace tranquilamente, recuerda desactivar la opción para mantener la seguridad de navegación.

Una vez solucionado el problema, al abrirlo en local debemos ver la siguiente imagen.

Fig. 2.3 Pantalla con textura cargada, una vez solucionado el error de seguridad.

INSTALAR UN SERVIDOR WEB LOCAL

Para poder probar las páginas web en local, y acercarnos más a la experiencia final del usuario, sin duda alguna la mejor opción es utilizar un servidor web. Para ello veremos algunas opciones disponibles que he dividido en dos posibilidades.

Una está dada por los servidores portables, que no requieren de instalación y es suficiente con ejecutarlos para disponer de un servidor listo para usar. Son una opción muy atractiva porque podemos invocarlos directamente desde un USB.

La segunda opción es instalar un servidor de forma permanente en el ordenador y así poder probar tranquilamente nuestras creaciones. Esta opción es la que nos da más funcionalidades y opciones de parametrización. Sin embargo, para usuarios no familiarizados, la configuración e instalación de un servidor suele ser costosa tanto en tiempo como en complejidad.

Servidores portables

PWS (Apache + MySQL + PHP)

PWS es un servidor web ligero y portable para *Windows* que destaca por su directa interfaz, donde con pocos clics podemos modificar las opciones de configuración y activar o desactivar módulos de *Apache* o *PHP*.

También incluye archivos de referencia, con documentación sobre tecnologías como *CSS3, JQuery, MySQL, PHP* o *Apache*.

La URL para descargarlo es http://sourceforge.net/projects/portableserver/

UwAmp (Apache + MySQL + PHP)

UwAmp es un servidor para Windows con una interfaz muy cómoda y útil. Además de las opciones comunes, ofrece un gráfico de estadísticas de consumo de *CPU* por servidor; unos gestores de configuración personalizados para *Apache*; *MySQL* y *PHP*; un administrador de bases de datos *SQLite,* así como aplicaciones de uso frecuente como *PHPMyAdmin* o XDebug. Ofrece soporte para múltiples versiones diferentes de *PHP*.

La URL para descargarlo es http://www.uwamp.com

Mongose

Mongose es un servidor portable disponible des del 2004. Al igual que los anteriores puedes abrirlo desde la carpeta donde está el código, copiando el fichero ejecutable. De esta forma dispones de un servidor sin necesidad de instalarlo ni de configurarlo. Puedes descargar el servidor desde los siguientes enlaces:

https://github.com/cesanta/mongoose
https://www.cesanta.com/products/binary

Servidores no portables.

XAMPP

Quizás uno de los más conocidos es *XAMPP*. Incorpora un servidor *Apache*, un sistema gestor de bases de datos *MySQL* y lenguajes como *PHP* y *Perl*. Además, ofrece soporte para gestionar cuentas *FTP*, acceso a bases de datos mediante *PHPMyAdmin*, bases de datos *SQLite* y otras características.

También incluye un servidor de correos *Mercury* para el envío de emails, un servidor *Tomcat* para *servlets JSP*, y un servidor *FTP FileZilla*.

Es multiplataforma, por lo que funciona en sistemas Windows, Linux, Mac e incluso hasta Solaris.

Puedes descargar el servidor en el siguiente enlace:

https://www.apachefriends.org

MAMP

Al igual que el anterior incorpora *Apache*, *MySQL*, *PHP*, *Perl*, y *Python*. La ventaja es su facilidad de instalación y desinstalación. *WAMP* se ofrece bajo la Licencia *GNU* (*General Public License*) y por lo tanto puede ser distribuido libremente dentro de los límites de esta licencia.

Dispones de una versión para *Windows* y otra para *Mac*. Puedes descargarlo en el siguiente enlace:

http://www.mamp.info

Web server para Node.js

La última propuesta que analizaremos es mediante *Node.js*. En caso de que dispongas en el ordenador de *Node.js* puedes utilizar el comando *"npm*" para instalar un servidor web sencillo.

Para configurar un servidor web local para *Node.js* es suficiente con seguir los pasos que listamos a continuación:

En primer lugar, debes instalar *node.js* desde https://nodejs.org en caso de que no dispongas de él. Puedes comprobar si ya está instalado en el ordenador ejecutando el comando ***npm*** desde la línea de comandos (símbolo del sistema en *Windows*). La salida que produce es similar al siguiente fragmento de texto:

```
Usage: npm <command>
```

```
where <command> is one of:
add-user, adduser, apihelp, ...
```

Ahora podemos instalar el servidor con el siguiente comando, para ello es necesario estar conectado a internet:

```
npm install -g http-server
```

Una vez finalizado, ya estamos listos para iniciar el servidor web mediante la siguiente línea de comandos:

```
http-server
```

Al ejecutar la instrucción aparecerá el siguiente texto:

```
Starting up http-server, serving ./
Available on:
http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

Ahora cualquier archivo contenido en la carpeta desde donde hemos ejecutado el comando es accesible a través de la URL http://127.0.0.1:8080. Podemos cerrar el servidor pulsando *CTRL-C*.

ESTADÍSTICAS (STATS.JS)

Cuando creamos aplicaciones complejas con *Three.js*, sobre todo juegos, con muchos objetos y animaciones, debemos vigilar que no sobrepasemos las limitaciones del sistema. Para ello puedes descargarte la librería stats.js https://github.com/mrdoob/stats.js/ que permite monitorizar los cuellos de botella, generando estadísticas de:

- **FPS** *Frames* renderizados por segundo. Este valor nunca debería bajar de 30, siendo aconsejable los valores cercanos a 60.
- **MS** Milisegundos necesarios para renderizar un *frame*. Cuanto más bajo sea el número mejor.
- **MB** Mbyte de memoria reservada. (Ejecuta Chrome con el parámetro ***--enable-precise-memory-info*** para poder obtener este tipo de estadísticas)

La librería, incluso permite definir nuestras propias estadísticas y está perfectamente integrada con *Three.js,* ya que su creador es el mismo que inició originalmente el proyecto de *Three.js*.

Para usar la librería basta con incluirla con la siguiente instrucción: **<script src="//rawgit.com/mrdoob/stats.js/master/build/stats.min.js"></script>**. Si la has descargado sustituye la URL por la dirección correcta. Puedes ver un ejemplo de su uso en:

https://www.thefiveplanets.org/b01/c02/02-statistics.html.

En el ejemplo hemos definido una función "*createStats*" que crea un recuadro con las estadísticas en la posición superior izquierda de la pantalla.

```
function createStats() {
    var stats = new Stats();
    stats.setMode(0); // 0: fps, 1: ms, 2: mb, 3+: custom

    return stats;
}
```

Con ***stats.setMode*** indicamos el tipo de estadísticas a presentar por defecto (0 - Frames renderizados por segundo, 1 - Milisegundos necesarios para renderizar un frame y 2 - Mbyte de memoria usada). La función la llamamos desde el método "*Init*", donde asignamos el objeto a una variable global *stats*, para poderlo referenciar desde otras partes del JavaScript.

```
stats = createStats();
```

Cada vez que volvemos a pintar la escena debemos forzar la actualización de las estadísticas con la llamada ***update***.

```
function render() {
    renderer.render(scene, camera);
    ....
    requestAnimationFrame(render);

    stats.update();
}
```

En la figura *2.4* vemos el recuadro de las estadísticas que se muestra al ejecutar el ejemplo. Podemos ir alternando de estadísticas haciendo *click* sobre el recuadro.

Fig. 2.4 Cuadro con las estadísticas.

CONTROL UI

Cuando estamos desarrollando, a menudo es necesario ajustar los valores de algunas variables para una visualización óptima. Es posible que queramos cambiar la escala, orientación o color de un objeto para una mejor visualización; o que simplemente queramos experimentar con las propiedades que nos ofrece *Threeе.js*. Para ello bastaría ir editando los valores de las variables en el código fuente y volver a cargar los archivos HTML. Este proceso puede ser tedioso y lento.

Por ello, podemos usar la librería *dat-gui*, para crear una interfaz gráfica que nos permita modificar las variables usadas para dibujar la escena de *Three.js* rápidamente. Puedes descargarte la última versión de la librería des de:

https://github.com/dataarts/dat.gui

En el siguiente enlace vemos un ejemplo básico, donde en la parte superior derecha hemos añadido una caja con tres variables que podemos alterar para modificar el comportamiento de la escena (la opacidad y el color del cubo, y la velocidad de rotación de la cámara):

https://www.thefiveplanets.org/b01/c02/03-controls.html.

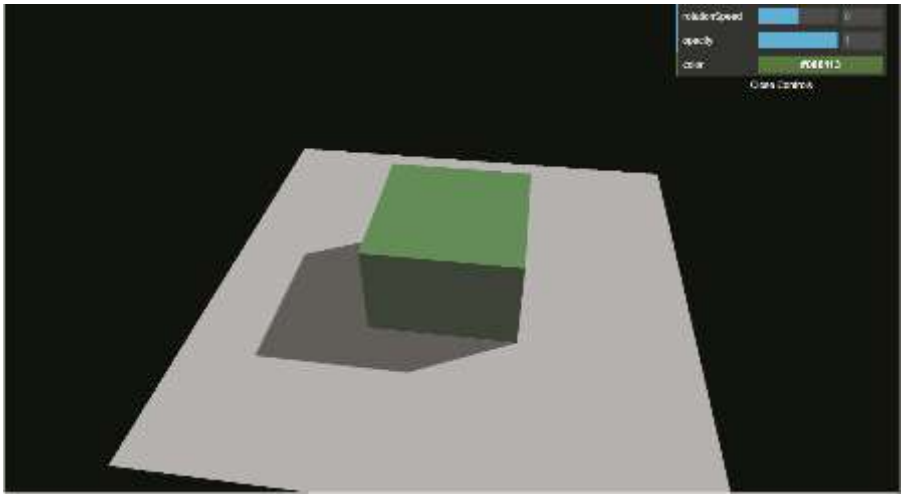

Fig. 2.5 Ejemplo de la incorporación de una Interface de Usuario realizada con dat-gui

Si analizamos el ejemplo vemos que, en primer lugar, en la parte superior hemos incluido la referencia al JavaScript descargado:

```
<script src="dat.gui.min.js"></script>
```

En segundo lugar, hemos creado un objeto con las variables que queremos controlar, y sus valores iniciales. El objeto lo hemos asignado a una variable global para poder acceder desde cualquier punto del código JavaScript.

```
var control;
function init() {
    ...
    control = new function() {
      this.rotationSpeed = 0.005;
      this.opacity = 0.6;
      this.color = 0x086113;
    };
    addControlGui(control);
}
```

En tercer lugar, hemos definido la función *addControlGui*, donde generamos la interfaz de usuario y vinculamos cada propiedad con el correspondiente campo.

```
function addControlGui(controlObject) {
    var gui = new dat.GUI();
    gui.add(controlObject, 'rotationSpeed', -1, 1);
    gui.add(controlObject, 'opacity', 0.1, 1);
    gui.addColor(controlObject, 'color');
}
```

Fíjate que al crear un nuevo campo (*gui.add*) hemos especificado cuatro argumentos.

El primer argumento es el objeto JavaScript que contiene las variables. En nuestro caso, es el objeto que se pasa a la función *addControls*. El segundo argumento es el nombre de la variable a añadir que debería apuntar a una de las variables (o funciones) disponibles en el objeto que se proporciona en el primer argumento. El tercer argumento es el valor mínimo que se debe mostrar en la interfaz gráfica de usuario. El último argumento especifica el valor máximo que se debe mostrar.

Ahora que ya disponemos de una interfaz gráfica de usuario que se puede usar para controlar las variables, lo único que queda para realizar en el bucle de renderizado de la escena es que se consulten los valores para actualizar la escena.

```
function render() {
    var delta =  clock.getDelta();
    var rotSpeed = delta*control.rotationSpeed;
    camera.position.x= camera.position.x * Math.cos(rotSpeed)
      + camera.position.z * Math.sin(rotSpeed);
    camera.position.z = camera.position.z * Math.cos(rotSpeed)
      - camera.position.x * Math.sin(rotSpeed);
    camera.lookAt(scene.position);
    var cube= scene.getObjectByName('cube').material
    cube.material.opacity = control.opacity;
    cube.material.color = new THREE.Color(control.color);
     requestAnimationFrame(render);
     renderer.render(scene, camera);
}
```

La interfaz gráfica podemos ocultarla o mostrarla pulsando la tecla H.

Tipos de campos a usar en la interfaz

En el ejemplo sólo hemos visto un conjunto pequeño de campos disponibles. La librería se adaptará al tipo de controlador según el tipo de variable que agregamos, o los distintos parámetros que indiquemos. Vemos algunos ejemplos a continuación.

- Para añadir un campo tipo lista desplegable para escoger un texto:

```
gui.add (controlObject, 'message',
 [ 'cube', 'sphere', 'plane' ] );
```

- Para añadir un campo tipo lista desplegable para seleccionar valores:

```
gui.add(controlObject,'speed',
{ Stopped: 0, Slow: 0.1, Fast: 5 } );
```

- Para especificar números con máximos y mínimos, o el valor de incremento:

```
// Incremento
gui.add(controlObject, 'noiseStrength').step(5);
// Mínimo y máximo
gui.add(controlObject, 'growthSpeed', -5, 5);
// Mínimo y incremento
gui.add(controlObject, 'maxSize').min(0).step(0.25);
```

- Para crear un botón con un script ejecutable:

```
controlObject.showmessage=function () { alert('message')}
// Min and max
gui.add(controlObject, 'growthSpeed', -5, 5);
```

- Para crear un cuadro de color, disponemos de la función *addColor*. Fíjate que en el ejemplo cambia el comportamiento según el tipo de variable que indiquemos, ya que el color lo podemos representar de varias formas (como un *string*, como un *array* de 3 o 4 números, o como un objeto).

```
var controlObject = new function() {
    // Cadena CSS
    this.color0 = "#ffae23";
    // RGB array
    this.color1 = [ 0, 128, 255 ];
    // RGB con opacidad
    this.color2 = [ 0, 128, 255, 0.3 ];
    // Matiz, saturation, valor
    this.color3 = { h: 350, s: 0.9, v: 0.3 };
};
var gui = new dat.GUI();

gui.addColor(text, 'color0');
gui.addColor(text, 'color1');
gui.addColor(text, 'color2');
gui.addColor(text, 'color3');
```

Fig.2.6 Ejemplo de selector de color.

- Para una casilla de verificación basta pasar como parámetro una variable de tipo booleano.

```
controlObject.visible=true;
gui.add(controlObject, 'visible');
```

Carpetas

Para facilitar el acceso a las propiedades puedes añadirlas dentro de carpetas que pueden plegarse o desplegarse como se ve en la figura *2.7*.

Fig.2.7 Ejemplo de carpetas anidadas.

```
var gui = new dat.GUI();

var f1 = gui.addFolder('Flow Field');
f1.add(controlObject, 'speed');

var f2 = gui.addFolder('Letters');
f2.add(controlObject,'growthSpeed');
f2.add(controlObject,'maxSize');
f2.add(controlObject,'message');
f2.open();
```

Eventos

Además de lo visto, se pueden añadir eventos que se disparan al momento de realizar el cambio de valor de una variable. Veamos un ejemplo a continuación:

```
var controller = gui.add(controlObject, 'maxSize', 0, 10);

controller.onChange( function(value) {
  /* Se dispara en cada cambio de valor, pulsación de Tecla, etc.*/
});

controller.onFinishChange( function(value) {
 /* Se dispara cuando pierdes el foco.*/
 alert("El valor es " + value);
});
```

DETECTAR SOPORTE WEBGL

La mayoría de los navegadores más modernos soportan *WebGL*, pero las versiones antiguas o de algunos dispositivos no lo soportan. Por ello es una buena idea asegurarse de que el navegador soporta *WebGL* al momento de crear

una instancia de *THREE.WebGLRender*. En caso de no soportarlo, la experiencia puede verse disminuida al aparecer errores JavaScript y una pantalla vacía. Una forma de verificarlo es con la función que adjuntamos a continuación.

Puedes ver un ejemplo en el siguiente enlace: https://www.thefiveplanets.org/b01/c02/04-detect-webgl-support.html.

```
function detectWebGL() {
    var testCanvas = document.createElement("canvas");
    var gl = null;
    try {
        gl = testCanvas.getContext("webgl",{antialias: false, depth: false });
    } catch (x) {
        gl = null;
    }
    if (gl==null) {
            try {
              gl = testCanvas.getContext("experimental-webgl", {antialias: false, depth: false });
            } catch (x) {
              gl = null;
            }
    }
    if (gl) {
        return true;
    } else {
        return false;
    }
}
```

ISBN EDICIÓN PAPEL: **978-15-399-0523-3**

ISBN EDICIÓN ELECTRONICA: **978-84-617-5141-9**

Más información en http://www.TheFivePlanets.org

Los navegadores web a lo largo de los años han ido incorporando nuevas tecnologías, convirtiéndose de simples visores de páginas con texto plano y alguna imagen a plataformas que permiten la creación de juegos en tres dimensiones.

La revolución empezó con la mejora de los estilos CSS, para continuar con la incorporación de bases de datos relacionales y documentales, y la posibilidad de crear aplicaciones que funcionan en modo offline sin conexión a internet. Los avances han continuado con la incorporación de tecnologías audiovisuales, la posibilidad del visionado de películas, la gestión del audio de forma nativa, incluso el reconocimiento de voz y la síntesis de voz para la lectura de textos. Finalmente, la incorporación de la tecnología 3D (WebGL) con aceleración por hardware han hecho que la programación para WEB no tenga nada que envidiar al resto de plataformas y lenguajes tradicionales.

Por todo ello, he decidido empezar una colección de libros que nos permita explorar estos avances y convertirnos en verdaderos expertos. Para distinguir la colección del resto de libros existentes en el mercado me he centrado en enseñar cómo realizar un juego de rol en primera persona, únicamente como excusa para introducir las tecnologías WEB paulatinamente.

¿Cómo es el juego?

El juego abarca desde la creación de las escenas en 3D, hasta la gestión de la música y efectos sonoros, la inteligencia artificial, la animación de los personajes y monstruos, el guardado de la partida en curso y su recuperación posterior y finalmente el empaquetado.

¿Qué cubre el primer libro de la colección?

El libro está dirigido a aquellos que desean explorar cómo crear animaciones 3D, ya sea para añadir elementos visuales a sus páginas WEB o de empresa, como aquéllos que quieran probar suerte en el diseño de juegos. En él aprenderemos a:

- Usar la librería THREE.JS para crear y animar una escena básica.
- A preparar el entorno de desarrollo y a solucionar los principales problemas que nos encontremos.
- A usar las geometrías y materiales que ofrece la librería por defecto.
- A crear texturas avanzadas y transparencias.
- A iluminar la escena y crear sombras.
- A cargar objetos realizados con terceras plataformas y animarlos con distintas técnicas.
- A usar el teclado y el ratón para controlar la cámara.

Finalmente crearemos nuestra primera localización del juego. Una bonita aldea medieval con su castillo y cementerio, llena de aldeanos con sus rutinas de paseo y quehaceres diarios, y un bosque espeso lleno de peligrosos monstruos.

¿Qué conocimientos previos son necesarios?

Este libro es adecuado para cualquiera con conocimientos básicos de JavaScript y HTML. No se necesitan conocimientos de matemática avanzada ni de WebGL. Todos los materiales y ejercicios pueden ser descargados libremente en la página *www.TheFIvePlanets.org*.

www.TheFivePlanets.org