Rodrigo Sánchez Valle

I.E.S. Francisco Romero Vargas

2ºCFGS Desarrollo de Aplicaciones Web

Jerez de la Frontera, Cádiz

Tabla de contenido

Presentación

Este juego puede ser jugado desde cualquier navegador web y sistema operativo. Para ello usaremos JavaScript desde el cliente, y para el servidor PHP, que está alojado en mi propio ordenador a través de XAMPP, que se encargará de guardar la partida y servir los recursos para el juego.

El juego es 2D, cuenta 2 cámaras diferentes. Una cámara cenital para mostrar las cosas desde arriba y nos permite explorar el mundo del juego. Sin embargo, cuando entremos a un nivel, la cámara cambia a una vista lateral para que nos permita explorar el nivel, a este tipo de juegos se le conoce como *Scroll lateral*.

Una de las funciones que tiene es poder guardar la partida en un servidor. De esta forma, el usuario tendrá siempre su partida guardada de forma online y podrá jugar desde cualquier ordenador entrando con su cuenta. A su vez, esto me permite incluir más niveles, objetos o personajes desde el servidor con solo actualizar unas pocas entradas en la base de datos. Así, la próxima vez que el usuario entre a la partida ya podrá ver las novedades sin tener que descargar o actualizar nada.

Bucle principal del juego

La función de esta parte del código en simple, y es hacer que JavaScript esté siempre abierto, ejecutándose.

La idea principal es que nuestro navegador se ejecute 60 veces por segundo para que la velocidad de dibujado sea lo suficientemente rápida. Para ello se usa el siguiente método:

```
var buclePrincipal =
{
    idEjecucion: null,
    ultimoRegistro: 0,
    aps: 0,
    fps: 0,
    iterar: function(registroTemporal)
    {
        buclePrincipal.idEjecucion = window.requestAnimationFrame(buclePrincipal.iterar);
        buclePrincipal.actualizar(registroTemporal);
        buclePrincipal.dibujar();

        if(registroTemporal - buclePrincipal.ultimoRegistro > 999)
        {
            buclePrincipal.ultimoRegistro = registroTemporal;
            console.log("APS: " + buclePrincipal.aps + " | FPS: " + buclePrincipal.fps);
            buclePrincipal.aps = 0;
            buclePrincipal.fps = 0;
```

El método **`window.requestAnimationFrame()`** informa al navegador que quieres realizar una animación y solicita que el navegador programe el repintado de la ventana para el próximo ciclo de animación. Haciendo esto se consigue hacer un Callback. Cuando se ejecuta esa parte del código, el valor que devuelve es el tiempo medido en milisegundos desde su ejecución, e inyecta ese valor en **`registroTemporal.`**

Dimensiones de la pantalla

Para conseguir que el juego esté bien posicionado en el navegador, hay que obtener el alto y el ancho del mismo, porque no siempre tendrá el mismo tamaño, ya que el usuario puede cambiar el tamaño o minimizar la ventana. Para poder conseguir las dimensiones, se usa la función **`addEventListener`**

```
var dimensiones =
{
    ancho: window.innerWidth || document.documentElement.clientWidth || document.body.clientWidth, //obtener ancho del navegador
    alto: window.innerHeight || document.documentElement.clientHeight || document.body.clientHeight, //obtener alto del navegador
    ladoTiles: 100,
    escala: 1,
    iniciar: function()
    {
        window.addEventListener("resize", function(evento)
        {
            dimensiones.ancho = window.innerWidth || document.documentElement.clientWidth || document.body.clientWidth;
            dimensiones.alto = window.innerHeight || document.documentElement.clientHeight || document.body.clientHeight;
        });
    },
    obtenerTilesHorizontales: function()
```

El evento que vamos a "escuchar" es **`resize`** para poder capturar el tamaño de la ventana cuando esta sufra una modificación y le podamos reasignar un valor a **`ancho`** y **`alto`**.

```
obtenerTilesHorizontales: function()
{
    var ladoFinal = dimensiones.ladoTiles * dimensiones.escala;
    return Math.ceil((dimensiones.ancho - ladoFinal) / ladoFinal);
},
obtenerTilesVerticales: function()
{
    var ladoFinal = dimensiones.ladoTiles * dimensiones.escala;
    return Math.ceil((dimensiones.alto - ladoFinal) / ladoFinal);
},
obtenerTotalTiles: function()
{
    return dimensiones.obtenerTilesHorizontales() * dimensiones.obtenerTilesVerticales();
}
```

Por último, estas tres funciones tienen una finalidad simple. Obtienen la cantidad de tiles que caben de izquierda a derecha según el tamaño de la pantalla.

Rectángulos

La clase **`Rectangulos`** tiene una finalidad muy precisa, y es la creación de rectángulos para nuestro juego. Básicamente, nuestro juego funcionará con rectángulos; el personaje, las localizaciones y las hitbox son, en esencia, son rectángulos.

La función **`.cruza`** devuelve **true** o **false** según si un rectángulo esta "cruzando" a través de otro. Si este devuelve true, significa que las figuras están chocando, si devuelve false, no están en contacto. Esto es fundamental a la hora de saber si el personaje está entrando en un nivel del mapa, o está colisionando con montañas, árboles o

Rectangulo.js M

```
function Rectangulo(x, y, ancho, alto, tipo)
{
    this.x = x;
    this.y = y;
    this.ancho = ancho;
    this.alto = alto;
    this.idHTML = tipo + "x" + x + "y" + y;
    this.html = '<div id="' + this.idHTML + '"></div>';
}

Rectangulo.prototype.cruza = function(rectangulo)
{
    return (this.x < rectangulo.x + rectangulo.ancho &&
        this.x + this.ancho > rectangulo.x &&
        this.y < rectangulo.y + rectangulo.alto &&
        this.alto + this.y > rectangulo.y) ? true : false;
}

Rectangulo.prototype.aplicarEstiloTemporal = function(colorHexadecimal)
{
    if (!document.getElementById(this.idHTML))
    {
        throw("El ID " + this.idHTML + " no existe en la hoja");
    }

    //var color = "#ff0000";
    document.getElementById(this.idHTML).style.backgroundColor = colorHexadecimal;

    document.getElementById(this.idHTML).style.position = "absolute";
    document.getElementById(this.idHTML).style.left = this.x + "px";
    document.getElementById(this.idHTML).style.top = this.y + "px";
    document.getElementById(this.idHTML).style.width = this.ancho + "px";
    document.getElementById(this.idHTML).style.height = this.alto + "px";
    document.getElementById(this.idHTML).style.zIndex = "5";
}

Rectangulo.prototype.mover = function(x, y)
{
    document.getElementById(this.idHTML).style.transform = 'translate3d(' + x + 'px,' + y + 'px, 0)';
}
```

cualquier entorno, ya sea en el mapamundi, o en el nivel en el que se encuentre.

En **.mover**, gracias a la función **translate3d()** es una función de que se utiliza para mover un elemento en un espacio tridimensional. Esta función nos viene muy bien para nuestro proyecto ya que tiene las funciones idóneas.

Detección de teclas

Como es de esperar, esta parte del código nos permite conocer qué tecla del teclado estamos pulsando, para posteriormente asignarle una función, como puede ser mover al personaje de arriba abajo en el minimapa, acceder a un nivel o saltar.

```
JS teclado.js > ...
1   var teclado =
2   {
3       teclas: new Array(),
4       iniciar: function()
5       {
6           document.onkeydown = teclado.guardarTecla;
7           document.onkeyup = teclado.borrarTecla;
8       },
9       guardarTecla: function(e)
10      {
11          if(teclado.teclas.indexOf(e.key) == -1)
12          {
```

En esta parte del código, lo más interesante es el uso de los eventos **.onkeydown** y **.onkeyup**. El primero se ejecuta cuando el usuario esté presionando la tecla, y el segundo evento se ejecutará cuando este suelte la tecla. Esto lo hacemos para detectar el movimiento del personaje hacia una dirección, y cuando el código detecta que el usuario ha soltado la tecla, hará que nuestro personaje se detenga donde sea.

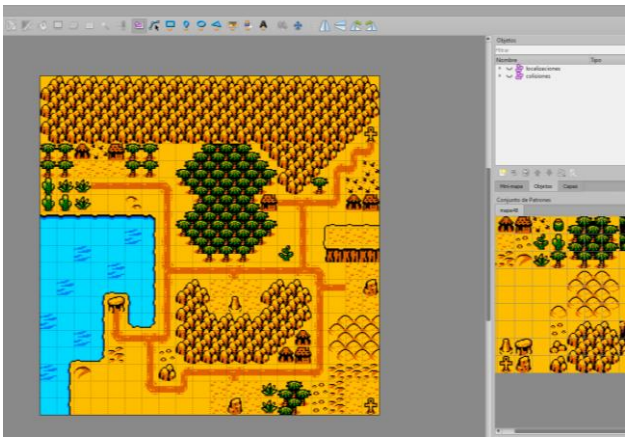
```
guardarTecla: function(e)
{
    if(teclado.teclas.indexOf(e.key) == -1)
    {
        teclado.teclas.push(e.key);
    }
},
```

```
borrarTecla: function(e)
{
    var posicion = teclado.teclas.indexOf(e.key);
    if(posicion !== -1)
    {
        teclado.teclas.splice(posicion, 1);
    }
},
```

Hojas de sprites y Tiled

Tiled

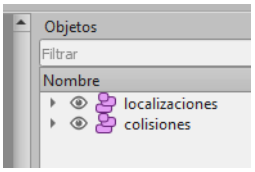
Para el diseño de niveles y el mapa, he decidido usar la aplicación Tiled. Esta herramienta tiene las características idóneas para este tipo de proyectos, ya que el dibujado es muy simple; solo tenemos que bajarnos una plantilla de sprites de internet (https://opengameart.org/) y a partir de esta, diseñar tu propio mapa.



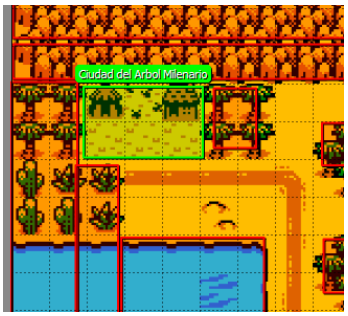
Hoja de sprites

Una herramienta indispensable es la de “seleccionar objetos”. Su función es la de generar un área delimitada por unas coordenadas registradas en un fichero *.json*, un identificador y un color para diferenciarlo todo visualmente.

Aquí podemos observar cómo, al poner visibles las capas de objetos, se nos muestra de forma visual dos tipos de áreas distintas, una verde y una roja. Se puede intuir a simple vista la función de cada tipo de área:



- **Colisiones**: los cuadrador rojos delimitan las zonas por las que nuestro personaje no podrá caminar, a esto se le denomina “hitbox”. Se les suele aplicar a zonas de agua no navegable, arboles, montañas, delimitando el mapa y cualquier zona a la que no deseemos que el jugador acceda .
- **Localizaciones**: los cuadrados verdes marcan las ubicaciones a las que el jugador puede acceder para entrar a un nivel. Estas tienen un nombre, que hace de identificador. Al pulsar una tecla especifica, el usuario accederá al nivel en cuestión.



Para finalizar, debemos exportar el fichero con dos formatos distintos: *json* y *png*. En el *json* podremos encontrar las distintas capas creadas, con las coordenadas en las que se crearon, su tamaño, su nombre y varias características del mismo, las cuales posteriormente usaremos. A demás de todo esto, hay muchos mas datos, como por ejemplos los *tilesets,* que son los datos propios del mapa: nombre, ruta, alto, anto, etc.

Cargar archivos sin caché

La función de este fichero tiene una finalidad muy definida, y es cargar los archivos *.js* necesarios para el juego sin necesidad de recurrir a la caché. La única “complicación”, por llamarlo de alguna manera, que he tenido con este fichero, ha sido el orden en el que se ordenan todos los *.js*, ya que hay algunos que dependen de otros, lo cual hace que se deban ejecutar antes que otros.

Uno de los motivos por los que me decidí a hacer este fichero es por el tema de los cambios en el código. La primera vez que visitas una web, el navegador lo lee y lo guarda en la memoria, la siguiente vez que entremos, como el navegador ya tiene una versión reciente, y hasta que no pase un cierto tiempo, este no considera necesario descargar de nuevo los archivos. Eso, para mí como desarrollador, es problemático, porque al hacer algún cambio en los ficheros y querer ver el cambio, al estar guardado el estado del navegador en caché,

tendría que recargar pulsando Ctrl+F5, pero por profesionalidad y por comodidad, decidí hacerlo de esta manera.

```
<div id="juego">_</div>
<script src="js/debug.js?1620933558"></script>
<br>
<script src="js/dimensiones.js?1620933558"></script>
<br>
<script src="js/EstadoPantallaTitulo.js?1620933558"></script>
<br>
<script src="js/inventario.js?1620933558"></script>
```

La forma en la que lo conseguí es sencilla. Solo tenia que añadir al nombre del fichero, un numero distinto cada vez que se recargue la pagina, para que el navegador considere que es necesario cargar en memoria todos los ficheros. Usé la función **`getTimestamp()`**para añadir una numeración distinta gracias a **`DateTime();`**

Sprites

Esta clase se encargará de administrar todas las capas de sprites del juego, o mejor dicho, en la hoja de sprites.

```
function Sprite(ruta, idSobreZero, posicionEnHoja)
{
    var elementosRuta = ruta.split("/");
    this.rutaHojaOrigen = "img/" + elementosRuta[elementosRuta.length - 1];
    this.idSobreZero = idSobreZero;
    this.idSobreUno = idSobreZero + 1;
    this.posicionEnHoja = posicionEnHoja;
}
```

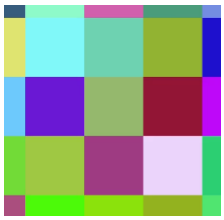
En primer lugar, necesitamos reconstruir la ruta, porque en el *.json* de nuestro mapa, la ruta aparece de esta manera tan "peculiar". Separamos la cadena cada vez que se encuentra el carácter "/" y cogemos solo el último elemento para obtener el nombre del archivo.

```
"tilesets":[
        {
         "columns":17,
         "firstgid":1,
         "image":"..\/img\/mapa48.png",
         "imageheight":432,
         "imagewidth":816,
```

```
function Punto(x, y)
{
    this.x = x;
    this.y = y;
}

Punto.prototype.coincide = function(punto)
{
    return (this.x == punto.x && this.y == punto.y) ? true : false;
}
```

Punto se refiere a un punto en una rejilla de coordenadas. La función de esta clase es indicar si dos puntos coinciden o no. El uso del objeto **`Punto.prototype.coincide`** es muy importante, ya que esto hace que esta función ocupe menos espacio en memoria. Todas las instancias que vayamos a crear de *Punto* y objetos no van a tener su propia copia de este método, solo habrá una para todas, y como este método va a hacer lo mismo en todos los casos, nos va a ahorrar un montón de memoria.

Tiles



Los tiles son cada uno de estos cuadraditos que nosotros hemos generado anteriormente, y cada uno de ellos son un cuadradito del mapa que contiene su propio Sprite.

```
function Tile(xEnTiles, yEnTiles, z, ancho, alto, sprite)
{
    this.rectangulo = new Rectangulo(xEnTiles, yEnTiles, ancho, alto);
    this.zIndex = z;
    this.sprite = sprite;
    this.idHTML = "x" + xEnTiles + "y" + yEnTiles + "z" + z;
    this.html = '<div id="' + this.idHTML + '"></div>';
}

Tile.prototype.aplicarEstilos = function()
{
    if (!document.getElementById(this.idHTML))
    {
        throw("El ID " + this.idHTML + " no existe en la hoja");
    }

    document.getElementById(this.idHTML).style.position = "absolute";
    document.getElementById(this.idHTML).style.left = (this.rectangulo.x * this.rectangulo.ancho) + "px";
    document.getElementById(this.idHTML).style.top = (this.rectangulo.y * this.rectangulo.alto) + "px";
    document.getElementById(this.idHTML).style.width = this.rectangulo.ancho + "px";
    document.getElementById(this.idHTML).style.height = this.rectangulo.alto + "px";
    document.getElementById(this.idHTML).style.zIndex = "" + this.zIndex;
    document.getElementById(this.idHTML).style.background = "url('" + this.sprite.rutaHojaOrigen + "')";

    var x = this.sprite.posicionEnHoja.x;
    var y = this.sprite.posicionEnHoja.y;

    document.getElementById(this.idHTML).style.backgroundPosition = "-" + x + "px -" + y + "px";
    document.getElementById(this.idHTML).style.backgroundClip = "border-box";
    document.getElementById(this.idHTML).style.outline = "1px solid transparent";
}
```

`this.rectangulo` va a ser la posición del Tile, el rectángulo que lo va a representar.`this.idHTML` es la id del elemento HTML que representará el tile en el navegador

El resto del código simplemente le da un estilo a los Tiles en píxeles.

Paleta de sprites

En su medida, nuestra paleta de sprites se puede parecer a la paleta de colores de un pintor, que es donde tiene todos los colores para su dibujo. En nuestro caso, la paleta de sprites va a contener todos los sprites necesarios para nuestro mapa.

Los **`datosSprites`** que recibe nuestra función vienen del archivo ***.json*** del mapa creado para el juego.

La siguiente parte del código se encarga de la parte más interesante de la función, que es guardar los sprites:

```
17      this.sprites = [];
18
19      for (s = 0; s < this.totalSprites; s++)
20      {
21          var idActualSobreZero = this.primerSpriteSobreUno - 1 + s;
22          this.sprites.push(new Sprite(this.rutaImagen, idActualSobreZero,
23              this.obtenerPosicionDesdeIdSprite(idActualSobreZero)));
24      }
```

Hacemos un bucle del total de sprites, que contiene el ancho de la imagen medido en sprites multiplicado por el alto de la imagen medido en sprites.

Se crea la variable **`idActualSobreZero`** para hacer que el id del Sprite, en vez de ser 1, sea 0, ya que Tiled tiene el "inconveniente" de que el primer id de la lista de sprites es 1.

Continuamos rellenando el array de sprites con la función **`.push()`** , creamos un nuevo Sprite y le pasamos los argumentos necesarios para crearlo; la ruta, el id sobre cero y la posición en la hoja.

Capa de tiles

La capa de tiles consiste en, valga la redundancia, las diferentes capas que va a tener nuestro juego web. Una primera capa sería el fondo, o sea, el mapa, por encima puede estar los elementos que no se pueden atravesar, como muros o montañas, las localizaciones donde el personaje puede acceder, y como última capa, el propio personaje. En este proyecto, se desarrolló con una única capa, para simplificar el trabajo, pero no se descarta que en un futuro se modifique el código para hacerlo funcionar en varias capas.

```
for (y = 0; y < this.altoEnTiles; y++)
{
    for (x = 0; x < this.anchoEnTiles; x++)
    {
        var idSpriteActualSobreUno = datosCapa.data[x + y * this.anchoEnTiles];
        if (idSpriteActualSobreUno == 0)
        {
            this.tiles.push(null);
        }
        else
        {
            var spriteActual = this.encontrarSpriteEnPaletaPorId(idSpriteActualSobreUno - 1, paletasSprites);

            this.tiles.push(new Tile(x, y, indiceZ, anchoDeLosTiles, altoDeLosTiles, spriteActual));
        }
    }
}
```

Recorremos los 20 tiles verticales de arriba hacia abajo con el bucle externo, y con el interno, los 20 tiles horizontales de izquierda a la derecha. Lo que leeremos aquí es este array de nuestro mapa *.json*, una coordenada, para saber que Sprite tenemos que seleccionar.

Como estamos trabajando con una única paleta de sprites, se hace un bucle de la paleta de sprites para leer cada una de ellas. Accedemos a la paleta actual y obtenemos el valor del primer Sprite para saber cuál es la paleta de sprites con la que estamos trabajando. En programación, eso se saca de la siguiente forma:

```
CapaMapaTiles.prototype.encontrarSpriteEnPaletaPorId = function(idSpriteSobreZero, paletasSprites)
{
    for (s = 0; s < paletasSprites.length; s++)
    {
        if (idSpriteSobreZero >= paletasSprites[s].primerSpriteSobreUno - 1 &&
            idSpriteSobreZero < paletasSprites[s].totalSprites + paletasSprites[s].primerSpriteSobreUno + 1)
            {
            return paletasSprites[s].sprites[Math.abs(paletasSprites[s].primerSpriteSobreUno - 1 - idSpriteSobreZero)];
        }
    }
    throw "El ID sobre ZERO " + idSpriteSobreZero + " del sprite no existe en ninguna paleta";
}
```

Máquina de Estados

Es el mecanismo que se va a encargar de alternar los estados del juego. Algunos de sus estados son: visualizando el mapa del mundo, dentro del nivel, mientras el juego carga, el menú del juego…

Crearemos un fichero para crear una constante para listar el estado de nuestro juego.

Después, crearemos el fichero importante, el de máquina de estados. Este controlará en qué estado está el juego actualmente y qué es lo que se está ejecutando.

```
var maquinaEstados =
{
    estadoActual: null,
    iniciar: function()
    {
        maquinaEstados.cambiarEstado(listadoEstados.PANTALLA_TITULO);
    },
    cambiarEstado: function(nuevoEstado, objetoEntradaLocalizacion)
    {
        switch(nuevoEstado)
        {
            case listadoEstados.CARGANDO:
                break;
            case listadoEstados.MENU_INICIAL:
                break;
            case listadoEstados.MAPAMUNDI:
                maquinaEstados.estadoActual = new EstadoMapamundi(listadoEstados.MAPAMUNDI, "mapas/desierto48.json", 500, 500);
                break;
            case listadoEstados.NIVEL:
                maquinaEstados.estadoActual = new EstadoMapamundi(listadoEstados.NIVEL, objetoEntradaLocalizacion.rutaMapa,
                    objetoEntradaLocalizacion.coordenadaXInicial, objetoEntradaLocalizacion.coordenadaYInicial);
                    console.log("Nivel 1");
                break;
            case listadoEstados.PANTALLA_TITULO:
                console.log("Iniciando pantalla");
                maquinaEstados.estadoActual = new EstadoPantallaTitulo();
                break;
        }
    },
    actualizar: function(registroTemporal)
    {
        maquinaEstados.estadoActual.actualizar(registroTemporal);
    },
    dibujar: function()
    {
        maquinaEstados.estadoActual.dibujar();
    }
}
```

Nada más iniciarlo, cambiaremos su estado y elegimos el primer estado el que queremos que aparezca, en este caso **PANTALLA_TITULO.**

Al **listadoEstados.MAPAMUNDI,** en resumen, la añadimos esa línea para darle una identificación, además de los atributos necesarios para crearlo. Lo mismo sucede con el estado **listadoEstados.NIVEL** y el estado **listadoEstados.PANTALLA_TITULO**

Tendremos una función llamada **cambiarEstado,** que evaluará mediante un switch el estado del juego.

Mapa

El archivo del mapa es el fichero más grande y contiene funciones algo más complejas. Gracias a este, podremos controlar tanto los mapas cenitales como los laterales.

Para poder entender bien el mecanismo de esta parte del proyecto, vayamos enumerando cada parte del mismo

this.posicion = new Punto(0,0);	Será 0,0 porque gracias a CSS, podemos mover el mapa entero de un lado a otro, y así conseguiremos que no tenga mucho impacto en el rendimiento,
this.posicionActualizada = new Punto(0,0);	Sirve para distinguir si en cada actualización el mapa debe moverse o no, ya que, si no debe moverse, es inútil estar llamando al

	método que lo va a mover o dibujar, porque eso desperdicia recursos.

this.anchoMedidoEnTiles = parseInt(objetoJSON.width);
this.altoMedidoEnTiles = parseInt(objetoJSON.height);
this.anchoDeLosTiles = parseInt(objetoJSON.tilewidth);
this.altoDeLosTiles = parseInt(objetoJSON.tileheight);
	Obtenemos el string que viene del fichero json y cogemos el atributo, y como viene en formato de texto, hay que transformarlo a numero

this.iniciarCapas(objetoJSON.layers);	Seleccionamos las capas que necesitamos del fichero json

Estado del Mapamundi

El propósito de esta clase es controlar la ejecución del juego mientras se está en el mapamundi, es decir, mientras el jugador explora el mapa del mundo para acceder a los niveles.

```
function EstadoMapamundi(idEstado, rutaMapaJSON, xInicial, yInicial)
{
    var that = this;
    this.mapaListo = false;
    this.mapa = null;
    this.nivel = null;
    this.jugadorMapamundi = null;
    this.jugadorNivel = null;
    ajax.cargarArchivo(rutaMapaJSON, function(objetoJSON)
    {
        that.mapa = new Mapa(objetoJSON, idEstado);
        that.mapaListo = true;
        that.jugadorMapamundi = new JugadorMapamundi
            (new Punto(xInicial, yInicial), idEstado);
        console.log("mapa cargado por AJAX");
    });
}
```

De las partes más interesantes del código de este fichero es, **`ajax.cargarArchivo()`** de forma asíncrona. Le tendremos que pasar una ruta y una función anónima donde admitiremos un objeto *.json*, y dentro de esto tenemos que coger los datos y dárselos al mapa. Dentro, simplemente tenemos que coger los datos y dárselos al mapa. El problema viene cuando dentro de la función anónima, al ser un bloque de código totalmente independiente al resto, no puedo acceder, por ejemplo, al *.mapa*. Así que decidí hacer un "arreglo", que consiste en crear una **`var that = this;`**. Gracias a esto, desde dentro de la función anónima podremos acceder al **`EstadoMapamundi()`**. Ahora si podremos hacer referencia al objeto **`EstadoMapamundi()`**, a su atributo **`mapa`**, por ejemplo.

Gracias a esto último que se ha desarrollado, ya se vería el mapamundi de nuestro juego web.

Inicio

Crearemos un array de funciones.

Iniciadores

```
var inicio =
{
    iniciadores:
    [
        dimensiones.iniciar(),
        maquinaEstados.iniciar(),
        teclado.iniciar(),
        mando.iniciar(),
        buclePrincipal.iterar()
    ],
```

En **`iniciadores`** indicaremos que queremos que se inicie y en qué orden, y a demás estamos haciendo que se ejecuten una detrás de la otra, es decir, hasta que no se ejecute la primera y haya acabado, no comenzará la segunda, y así sucesivamente.

Iniciar Juego

Lo interesante de esta función es que haremos un Callback. Callback significa que podremos pasar una función como argumento a otra función, de esta forma la primera función ejecutará su código y cuando termine, se ejecutará detrás la segunda función.

```
iniciarJuego: function()
{
    inicio.encadenarInicios(inicio.iniciadores.shift());
},
```

En **`iniciarJuego`** llamaremos a **`encadenarInicios`**, le pasamos un iniciador como argumento, así

que le pasamos **inicio**, le pasamos nuestro arraya **`iniciadores`** y usamos la función de JavaScript **`.shift()`**. Shift devuelve el primer elemento y luego lo borra del array, cada vez que lo ejecutemos irá borrando uno a uno los elementos del array hasta que este esté vacío.

Encadenar Inicios

Como es lógico, una vez ejecutada y finalizada la función **`iniciarJuego()`**, eventualmente no devolveremos nada porque los elementos de **`iniciadores`** habrán sido borrados, y esto puede causar una excepción. Lo primero que tenemos que comprobar si el iniciador existe.

Necesitamos seguir usando la mecánica del Callback para hacer que una función llame a la siguiente.

```
encadenarInicios: function(iniciador)
{
    if(iniciador)
    {
        iniciador(() => inicio.encadenarInicios(iniciadores.shift()));
    }
}
```

Básicamente, este **primer bloque** está pasando una función como Callback, que se ejecutará cuando el **`iniciador`** termine, así que, por ejemplo, cuando **`maquinaEstados.iniciar()`** termine, se ejecutará el Callback que hay en el argumento. Esta sintaxis nos permite colocar una función anónima "()" y la flecha "=>" indica que **`iniciador`** devuelve como argumente lo que se ejecuta a la derecha de esta flecha. En resumen, es una función que se llama así misma.

Controles

```
var controles =
{
    arriba: false,
    abajo: false,
    izquierda: false,
    derecha: false,
    actualizar: function()
    {
        if (teclado.teclaPulsada(controlesTeclado.arriba))
        {
            controles.arriba = true;
        }
        if (teclado.teclaPulsada(controlesTeclado.abajo))
        {
            controles.abajo = true;
        }
        if (teclado.teclaPulsada(controlesTeclado.izquierda))
        {
            controles.izquierda = true;
        }
        if (teclado.teclaPulsada(controlesTeclado.derecha))
        {
            controles.derecha = true;
        }
    },
    reiniciar: function()
    {
        controles.arriba = false;
        controles.abajo = false;
        controles.izquierda = false;
        controles.derecha = false;
    }
};
```

```
var controlesTeclado =
{
    arriba: 'w',
    abajo: 's',
    izquierda: 'a',
    derecha: 'd',
    entrarLocalizacion: 'r',
    saltar: ' '
};
```

```
teclaPulsada: function(codigoTecla)
{
    return (teclado.teclas.indexOf(codigoTecla) !== -1) ? true : false;
}
```

Esta parte de código es relativamente simple. Tendremos dos funciones: actualizar y reiniciar. Antes de seguir con los controles, necesitaremos una clase intermedia para que traduzca los controles a teclas. Se puede observar en **`controlesTeclado`** las teclas asignadas a cada control, suyo sistema de navegación del personaje será el clásico WASD.

Una vez tengamos los controles traducidos, ya podemos empezar con **`controles.`** El método **`actualizar()`** se encargará de detectar si cierto control es verdadero o no. Usaremos la función **`teclaPulsada()`**, que está definida en el fichero **`teclado.js`**. Esta función simplemente nos dice si la tecla está pulsada o no. Despues, le indicamos que tecla queremos averiguar si ha sido pulsada. Por ultimo, la función reiniciar se va a encargar de poner denuevo en "false" todos los controles, para que no se quede el personaje llendo hacia una dirección constantemente y se pueda reiniciar el movimiento.

Mostrar al jugador en el mapa

A continucación, vamos a proceder a mostrar a nuestro protagonista en el mapa del juego. Al ser un fichero muy extenso (de casi 250 líneas de código), nos centraremos en comentar y explicar las partes mas interesantes del mismo.

Comenzaremos creando las propiedades fundamentales, como el alto y el ancho del cuadrado que ocupará, ruta de la imagen, origen del sprite en ambos ejes, velocidad de movimiento, etc.

Comencemos con lo mas interesante, las funciones. La pirmera que vamos a crear, será de **`JugadorMapamundi.prototype.aplicarEstilos`**. Aquí le daremos las reglas CSS necesarias para poder mostrar al personaje en el mapamundi. No nos detendremos mucho aquí, asi que explicaré de forma rapida lo que hace:

```
JugadorMapamundi.prototype.aplicarEstilos = function()
{
    var idHTML = "jugador";
    document.getElementById(idHTML).style.position = "absolute";
    document.getElementById(idHTML).style.left = this.posicionCentrada.x + "px";
    document.getElementById(idHTML).style.top = this.posicionCentrada.y + "px";
    document.getElementById(idHTML).style.width = this.ancho + "px";
    document.getElementById(idHTML).style.height = this.alto + "px";
    document.getElementById(idHTML).style.zIndex = "10";
    document.getElementById(idHTML).style.background = "url('" + this.rutaHojaSprites + "')";
    document.getElementById(idHTML).style.backgroundPosition = "-" + this.origenXSprite + "px -" + this.origenYSprite + "px";
    document.getElementById(idHTML).style.backgroundClip = "border-box";
    document.getElementById(idHTML).style.outline = "1px solid transparent";
}
```

Muestra el pesonaje en una posición absoluta, le da las coordenadas en el eje X e Y en píxeles, un ancho y un alto, **zIndex** para que el jugador esté encima del mapa y nos asegurarnos de que no hay nada que lo tape, la ruta de la imagen del personaje, posición del mapamundi, backgroundClip para especificar el fondo hasta el borde y un borde transparente que delimita el mapa.

Para que esto empiece a funcionar, hay que añadir al fichero **`inicio.js`** es **`dimensiones.iniciar()`** para que calculen las dimensiones, y en **EstadoMapamundi()** denbajo de **`this.mapa`**, se tiene que icluir **`this.JugadorMapamundi = null`** porque el estado del mapamundi va a intentar va a intentar crear una instancia del objeto **`jugador`**.

Finalmente, dentro de **`ajax.cargarArchivo()`** se debe incluir la siguiente linea: **`that.jugadorMapamundi = new JugadorMapamundi(new Punto(xInicial, yInicial), idEstado);`** y gracias a esto conseguiremos mostrar a nuestro jugador en la posición del mapa deseada.

Mover el mapa

Vamos a proceder a mostrar a nuestro personaje en el mapamundi. Antes de nada, aclarar que realmente, el jugador no se moverá. Para dar la sensación de que el personaje se está moviendo por el mapamundi, iremos desplazando el mapa de fondo en la dirección contraria, es decir, si yo quiero que el personaje se mueva hacia la derecha, el mapa se desplazará hacia la izquierda.

```
JugadorMapamundi.prototype.moverEnMapamundi = function() {
    this.velocidadX = 0;
    this.velocidadY = 0;

    if(!this.colisionArriba && teclado.teclaPulsada(controlesTeclado.arriba)) {
        this.velocidadY += this.velocidadMovimiento;
    }
    if(!this.colisionAbajo && teclado.teclaPulsada(controlesTeclado.abajo)) {
        this.velocidadY -= this.velocidadMovimiento;
    }
    if(!this.colisionIzquierda && teclado.teclaPulsada(controlesTeclado.izquierda)) {
        this.velocidadX += this.velocidadMovimiento;
    }
    if(!this.colisionDerecha && teclado.teclaPulsada(controlesTeclado.derecha)) {
        this.velocidadX -= this.velocidadMovimiento;
    }

    this.posicionEnMapaEnPixeles.x += this.velocidadX;
    this.posicionEnMapaEnPixeles.y += this.velocidadY;
}
```

Comenzaremos por añadir al archivo **`JugadorMapamundi`** el método **`.moverEnMapamundi`** donde intentaremos manejar los controles del jugador de forma sencilla. Llamamos a la clase **`teclado.teclaPulsada`** para saber si cierta tecla está pulsada, comprobamos si **`controlesTeclado.`** y una dirección para saber si está pulsada esa dirección. Si todo eso es así, le sumaremos o restaremos la velocidad de movimiento del personaje a la velocidad del eje necesario.

Ahora iremos a **`EstadoMapamundi`** y añadiremos la siguiente línea de código: **`this.jugadorMapamundi.actualizar(registroTemporal, this.mapa);`** ya que no lo estábamos usando.

Como **`EstadoMapamundi`** está contenido dentro de **`maquinaEstado`** tendremos que darle registro temporal a la función **`actualizar`**, y como **`maquinaEstado`** a través de **`buclePrincipal`** ya podremos pasarle el registro temporal a **`maquinaEstados.actualizar()`**. Todo esto es para dejarlo conectado para el futuro, ya que no vamos a usar para nada esta variable aún.

Para que todo esto se refleje en el mapa, este mismo debe conocer la existencia de nuestro jugador. Para ello se lo debemos pasar por argumento **`this.mapa.actualizar(registroTemporal, this.jugadorMapamundi.posicionEnMapaEnPixeles);`** En resumen, le estamos dando al mapa el objeto punto que indica la posición del jugador.

Nos dirigimos a **`Mapa.js`** y haremos la ultima parte en la cual le añadiremos esto para que el mapa siga el movimiento que le indiquemos a través del teclado.

```
Mapa.prototype.actualizar = function(registroTemporal, posicionJugadorEnPixeles)
{
    this.posicion.x = posicionJugadorEnPixeles.x;
    this.posicion.y = posicionJugadorEnPixeles.y;
```

Hacemos que la posición X del mapa equivalga a la posición del jugador en píxeles de X, e igual con el eje Y. Esto nos permite cambiar la posición del mapa.

Aun no estaría funcionando, ya que el mapa carece todavía de la función de moverse. Aunque cambiemos su posición, no serviría de nada porque realmente no estamos dibujando en ningún sitio los cambios. Lo que necesitamos ahora es registrar los cambios que hay en el mapa 60 veces por segundo y seguir moviéndolo.

Nos dirigimos a **`Tile.js`** y creamos la función **`mover`** y añadimos:

```
Tile.prototype.mover = function(x, y)
{
    document.getElementById(this.idHTML).style.transform = 'translate3d(' + x + 'px,' + y + 'px, 0)';
}
```

Para que esto funciones, volvemos a **`Mapa.js`** y hacemos que todos los tiles del mapa utilicen su método **`mover`** a la vez.

Por último, dentro de **`buclePrincipal`** tenemos que dibujar la máquina de estados, así que tendremos que añadir este trozo de código:

```
},
dibujar: function(registroTemporal) {
    maquinaEstados.dibujar();
    buclePrincipal.fps++;
}
```

Colisiones básicas

El objetivo es que tanto nuestro personaje como el entorno, consigan una hitbox. La finalidad es que cuando dos elementos se toquen o se crucen, no se puedan atravesar, porque lógicamente nuestro jugador no puede atravesar las montañas, los árboles, el agua o el final del mapa.

Esta parte del proyecto me parece de un gran interés e importancia, asi que iremos enumerando poco a poco cada paso que vamos dando.

`Rectangulo.js`	

Debemos averiguar si dos rectángulos se cruzan. La condición será: *<<si la posición X del nuestro rectángulo es menor que la X del rectángulo que le pasamos + el ancho del rectángulo, & si la X de nuestro rectángulo + el ancho de nuestro rectángulo es mayor que la X del rectángulo que le pasamos, & si sucede lo mismo para el eje Y junto con el alto, devolverá true, si no es así, false>>*	`Rectangulo.prototype.cruza = function(rectangulo)`
	`{`
	`  return (this.x < rectangulo.x + rectangulo.ancho &&`
	`  this.x + this.ancho > rectangulo.x &&`
	`  this.y < rectangulo.y + rectangulo.alto &&`
	`  this.alto + this.y > rectangulo.y) ? true : false;`
	`}`

Centrar el mapa

Vamos a conseguir que nuestro mapa del mundo quede centrado en la pantalla. Para ello, nos vamos a **`jugadorMapamundi.js`** e incluimos **`this.posicionEnMapaEnPixeles = this.posicionCentrada;`** ya que tenemos esto:

```
this.posicionCentrada = new Punto(centroX, centroY);
```

Esto funciona. El mapa se ha movido debajo del jugador, pero con un pequeño problema, y es que han aparecido una especia de rejilla de pixeles en nuestro mapa. Esto ha sucedido porque los navegadores, al interpretar números decimales, intenta simular como se vería medio pixel, por ejemplo, y la forma de representarlo sería algo así. Para evitar esto, tenemos que impedir las dimensiones detecten decimales para el alto y el ancho de la ventana. Lo solucionaremos usando la función matemática **`.trunc()`**

```
var centroX = Math.trunc
    (dimensiones.ancho / 2 - this.ancho / 2);
var centroY = Math.trunc
    (dimensiones.alto / 2 - this.alto / 2);
```

Ahora necesitamos que nuestro jugador esté donde nosotros queremos, y no tenerlo centrado en el centro de la pantalla. Para ello, construimos un nuevo **`new Punto(centroX, centroY);`**

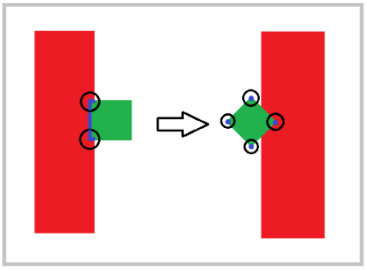
Como realmente estamos manejando "al revés" a nuestro personaje, para darle una coordenada inicial, debemos pensar en que los números están en positivo cuando realmente hay que escribirlos en negativo. Para evitar esto, crearemos este pequeño "truco".

```
posicionInicialEnPixeles.x *= -1;
posicionInicialEnPixeles.y *= -1;

this.posicionEnMapaEnPixeles = new Punto(this.posicionCentrada.x + posicionInicialEnPixeles.x,
    this.posicionCentrada.y + posicionInicialEnPixeles.y);
this.aplicarEstilos();
```

Límites y rectángulos

Necesitamos crear unos rectángulos para poder delimitar sus hitbox como ya hemos explicado anteriormente. Necesitaremos crear 4 zonas de colisión: arriba, abajo, izquierda y derecha. Por cada limite necesitaremos pasarle la coordenada X y la Y en la que se encuentras, el alto y el ancho. Como se puede ver en la captura, el ancho se divide entre 3 para darle solución a un problema. Si dejamos el ancho completo, o el alto según la posición de la que hablemos, conseguimos unas "colisiones pegajosas", es decir, el personaje, cuando entra en contacto con otra hitbox, no se puede mover a no ser que se le obligue a retroceder en la dirección contraria, no podrá desplazarse hacia otra dirección. A la izquierda podemos ver como personaje entra en contacto con una montaña y colisiona con 3 de sus 4 lados, o sea, que no se podría mover ni arriba, ni abajo y ni a la izquierda, solo a la derecha, y eso da la sensación de que tiene una hitbox "pegajosa". Al dividir el tamaño de la hitbox entre 3, hace que esta sea similar a un rombo, con una hitbox muy pequeña en el centro de cada lado, así conseguimos que, cuando entre en contacto con uno de los lados del personaje, este pueda tener acceso a moverse hacia las otras direcciones.

```
this.limiteArriba = new Rectangulo
    (centroX + this.ancho / 3, centroY, this.ancho / 3, 1);
this.limiteAbajo = new Rectangulo
    (centroX + this.ancho / 3, centroY + this.alto - 1, this.ancho / 3, 1);
this.limiteIzquierda = new Rectangulo
    (centroX, centroY + this.alto / 3, 1, this.alto / 3);
this.limiteDerecha = new Rectangulo
    (centroX + this.ancho - 1, centroY + this.alto / 3, 1, this.alto / 3);
```



```
this.limiteMapa = new Rectangulo(this.posicion.x,
    this.posicion.y,
    this.anchoMedidoEnTiles * this.anchoDeLosTiles,
    this.altoMedidoEnTiles * this.altoDeLosTiles, "colision");
```

A continuación, nos tenemos que dirigir a **Mapa.js** y crear la hitbox para el límite del mapa. Crearemos un nuevo rectángulo y le daremos los valores que necesita. Aquí surge una complicación, y es que cuando nosotros movemos el personaje, realmente lo que se desplaza es el mapa, pero no su límite. Así que debemos conseguir que esta también se mueva. Para ello, vamos a la función **actualizar** y en cada actualización vamos a colocar la colisión en el lugar donde se encuentre el mapa.

```
Mapa.prototype.actualizar = function(registroTemporal, posicionJugadorEnPixeles)
{
    this.posicion.x = posicionJugadorEnPixeles.x;
    this.posicion.y = posicionJugadorEnPixeles.y;

    this.limiteMapa.x = this.posicion.x;
    this.limiteMapa.y = this.posicion.y;
}
```

Límites del mapa

```
JugadorMapamundi.prototype.comprobarColisiones = function(mapa) {
    this.colisionArriba = false;
    this.colisionAbajo = false;
    this.colisionIzquierda = false;
    this.colisionDerecha = false;

    if (!this.limiteArriba.cruza(mapa.limiteMapa)) {
        this.colisionArriba = true;
    }
    if (!this.limiteAbajo.cruza(mapa.limiteMapa)) {
        this.colisionAbajo = true;
    }
    if (!this.limiteIzquierda.cruza(mapa.limiteMapa)) {
        this.colisionIzquierda = true;
    }
    if (!this.limiteDerecha.cruza(mapa.limiteMapa)) {
        this.colisionDerecha = true;
    }
```

Para crear el límite del mapa, tendremos que irnos a **EstadoMapamundi.js** y debemos crear la función **comprobarColisiones**. Esta parte del código tiene el trabajo de comprobar si el límite superior del personaje, por ejemplo, está cruzando la el límite del mapa. Si este es así, el valor de nuestra variable irá de variando según se cumpla la condición o no.

Aunque no lo parezca, esto ha sido más fácil de lo esperado, ya que teníamos creado ya la parte más importante del código necesario para crear esta función, como es establecer los límites del mapa, y la función **.cruza**.

Rectángulos y colisiones

```
Mapa.prototype.iniciarCapas = function(datosCapas)
{
    for (i = 0; i < datosCapas.length; i++)
    {
        if (datosCapas[i].name == "colisiones")
        {
            console.log("capa colisiones");
            for (c = 0; c < datosCapas[i].objects.length; c++)
            {
                this.rectangulosColisiones.push(new Rectangulo(
                    datosCapas[i].objects[c].x, datosCapas[i].objects[c].y,
                    datosCapas[i].objects[c].width, datosCapas[i].objects[c].height, "colision"
                ));
            }
        }
```

Para conseguir implementar las colisiones en el juego, vamos a trabajar con los ficheros **Mapa.js**, **Rectangulo.js** y **Tile.js.**

En **Mapa.js** creamos un array para guardar las colisiones de los rectángulos. creamos la función **iniciarCapas** donde leeremos una parte del código de nuestro fichero del mapamundi **desierto48.json**. Para leer el grupo de colisiones, tendremos que comprobar el nombre del objeto. Después tenemos que leer todos los rectángulos y crear los nuestros propios. Como podemos ver en la imagen, dentro de “objects” se encuentran todos nuestros rectángulos con los datos de cada uno de ellos. Ahora lo que debemos hacer es llamar a **this.rectangulosColisiones.push()** y dentro del array de colisiones, iremos creando nuestros rectángulos con todos los valores necesarios para su creación (coordenadas y tamaños).

```
"color":"#ff0000",
"draworder":"topdown",
"name":"colisiones",
"objects":[
        {
         "height":132,
         "id":1,
         "name":"",
         "rotation":0,
         "type":"",
         "visible":true,
         "width":954,
         "x":2,
         "y":2
        },
        {
         "height":41.3333,
         "id":2,
```

A continuación tendremos que insertar el HTML en las colisiones, para ello tendremos que ir a **Rectángulo.js** y añadir a la función **Rectangulo()** la parte del código marcada en esta imagen. Una vez hecho esto, podremos volver a **Mapa.js** y leer todos los datos necesarios y añadirlo al HTML.

```
function Rectangulo(x, y, ancho, alto, tipo) {
    this.x = x;
    this.y = y;
    this.ancho = ancho;
    this.alto = alto;
    this.idHTML = tipo + "x" + x + "y" + y;
    this.html = '<div id="' + this.idHTML + '"></div>';
}
```

```
var htmlColisiones = "";
for(c = 0; c < this.rectangulosColisiones.length; c++) {
    htmlColisiones += this.rectangulosColisiones[c].html;
}
document.getElementById("colisiones").innerHTML = htmlColisiones;
```

Para poder ver a simple vista todo esto que hemos hecho funciona perfectamente, añadiremos ciertos estilos a nuestro código para poder ver las colisiones en rojo y comprobaremos que no nos deja atravesarlas. Esto es algo temporal, asi que no hay nada relevante a la hora de explicar como se hizo esta parte del código.

Para finalizar, crearemos una función en **JugadorMapamundi.js** que nos permita comprobar todas las colisiones, tanto las colisiones de los "elementos rojos" como las del límite del mapa, la incluiremos todas.

Para ellos crearemos la función prototipada **.comprobarColisiones** que le pasaremos un objeto **mapa**, que podemos acceder a él gracias a que manejamos al jugador y al mapa, y como el objeto **jugadorMapamundi** tiene acceso a ambos, podemos "pasarlos entre ellos" sin ningún problema.

```
JugadorMapamundi.prototype.comprobarColisiones = function(mapa) {
    this.colisionArriba = false;
    this.colisionAbajo = false;
    this.colisionIzquierda = false;
    this.colisionDerecha = false;

    if (!this.limiteArriba.cruza(mapa.limiteMapa))
    {
        this.colisionArriba = true;
    }
    if (!this.limiteAbajo.cruza(mapa.limiteMapa))
    {
        this.colisionAbajo = true;
    }
    if (!this.limiteIzquierda.cruza(mapa.limiteMapa))
    {
        this.colisionIzquierda = true;
    }
    if (!this.limiteDerecha.cruza(mapa.limiteMapa))
    {
        this.colisionDerecha = true;
    }
```

Crearemos las variables booleanas de colisión arriba, abajo, izquierda y derecha, y las pondremos por defecto en **false** y haremos una serie de pruebas con distintos elementos y si no superan la condición, lo cambiaremos a **true**. Para esto usaremos la función **.cruza** que creamos anteriormente. Si el límite de uno de los lados **no** se está cruzando con el limite del mapa, el booleando de este lado pasará a ser **true**, mientras que se estén cruzando, estará en **false**.

Una vez esté todo comprobado, solo nos quedaría movernos en el mapamundi. Para ellos escribiremos la función **moverEnMapamundi** en nuestro fichero. Las condiciones sin sencillas, si no se está detectando la colisión, y la tecla está pulsada, la velocidad en el eje X o en el eje Y aumenta según sea necesario una u otra.

```
JugadorMapamundi.prototype.moverEnMapamundi = function() {
    this.velocidadX = 0;
    this.velocidadY = 0;

    if(!this.colisionArriba && teclado.teclaPulsada(controlesTeclado.arriba)) {
        this.velocidadY += this.velocidadMovimiento;
    }
    if(!this.colisionAbajo && teclado.teclaPulsada(controlesTeclado.abajo)) {
        this.velocidadY -= this.velocidadMovimiento;
    }
    if(!this.colisionIzquierda && teclado.teclaPulsada(controlesTeclado.izquierda)) {
        this.velocidadX += this.velocidadMovimiento;
    }
    if(!this.colisionDerecha && teclado.teclaPulsada(controlesTeclado.derecha)) {
        this.velocidadX -= this.velocidadMovimiento;
    }

    this.posicionEnMapaEnPixeles.x += this.velocidadX;
    this.posicionEnMapaEnPixeles.y += this.velocidadY;
}
```

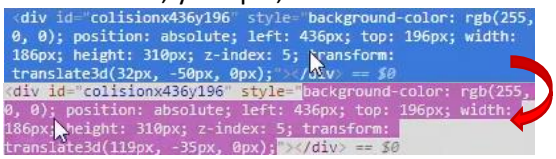
Por último, detectaremos si estamos en colisión o no con otro rectángulo. Iremos a la función **comprobarColisiones** y dentro de esta parte del código crearemos un bucle for para

```
for (var i = 0; i < mapa.rectangulosColisiones.length; i++) {
    var traduccionTemporalColision = new Rectangulo(
        mapa.rectangulosColisiones[i].x + mapa.posicion.x,
        mapa.rectangulosColisiones[i].y + mapa.posicion.y,
        mapa.rectangulosColisiones[i].ancho,
        mapa.rectangulosColisiones[i].alto
    );
```

circular por el mapa y por su objeto array **rectangulosColisiones**. En cada iteración del array, el rectángulo que haya en las colisiones traduciremos su posición de forma que concuerde con la del

jugador, ya que la forma en la que se lee la ubicación del personaje es distinta de la forma en la que se lee la de las localizaciones. Como se puede ver aquí, el jugador siempre está en un punto

concreto, porque mide la distancia en píxeles desde el borde de **nuestro navegador** hasta ese punto en el que se encuentra. Pero, los rectángulos de colisiones, está muy diferente por una razón sencilla, y es que, todas las colisiones del mapa miden su posición con respecto al borde

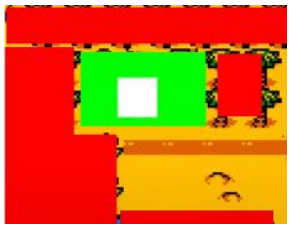


del mismo, así que se podría decir que no están hablando en el mismo idioma, y lo que necesitamos en un "traductor". De ahí la creación de la variable mostrada anteriormente: **`traduccionTemporalColision`**. Ahora, las colisiones se localizan respecto al jugador, no respecto al mapa.

En resumen, comprobamos que estamos dentro del mapa, comprobamos también que no haya rectángulos en medio y permitimos o no el movimiento.

Localizaciones

Para nuestro agrado, trabajar con las localizaciones es igual que con las colisiones, son zonas, partes del mapa, que tienen un área delimitada donde, si la hitbox del personaje hace contacto con la de la localización, en vez de colisionar, se debe atravesar, y con un botón del teclado, poder acceder al nivel deseado.



Una vez explicado esto, debemos crear un array de rectángulos para las localizaciones en **`Mapa.js`** para recoger las localizaciones, recorremos los datos de la capa para añadir los rectángulos de las localizaciones, creamos las localizaciones y las añadimos al HTML y, por último, las dibujamos.

Para poder mostrar su hitbox visualmente, se modificó el código para comprobarlo. Como podemos ver, las hitbox de las colisiones siguen estando rojas, pero las de las localizaciones ahora son verdes, y como se puede observar, nuestro personaje puede atravesarlas, ya que debe poder para acceder al nivel que contenga.

El siguiente paso es que cuando nuestro personaje toque la localización, aparezca una ventana emergente con el nombre del lugar, y así hay una referencia visual del nivel al que accedemos.

Sprite del personaje

Elegir bien nuestra paleta de sprites para el personaje es de las cosas mas importantes del juego, ya que va a ser la mayor representación de nuestro proyecto. En mi caso, he querido diseñar yo mismo al mismo.

Primero incluiremos la imagen, después queremos que al modificar la dirección del personaje su Sprite también cambie y, por último, vamos a ver la animación del personaje moviéndose.

```
this.rutaHojaSprites = "img/allFaces.png";

this.origenXSprite = 0;
this.origenYSprite = this.alto * this.personaje;

this.velocidadMovimiento = 2;

this.velocidadX = 0;
this.velocidadY = 0;

this.enMovimiento = false;
this.framesAnimacion = 0;
```

Para comenzar con este proceso, vamos a indicarle la ruta de la imagen, las coordenadas de los sprites que queremos utilizar. También queremos la velocidad a la que irá nuestro personaje, y la dirección a la que se dirige. Por último, necesitaremos saber si el jugador está en movimiento o no, y cuantos frames necesita para realizar la animación de andar.

Añadiremos para un uso posterior la línea siguiente **`this.rectanguloGeneral = new Rectangulo(centroX, centroY, this.ancho, this.alto);`** Su función será la de detectar la zona en la que se encuentra el personaje para que cuando estemos sobre una localización sepamos que la estamos tocando.

```
JugadorMapamundi.prototype.aplicarEstilos = function()
{
    var idHTML = "jugador";
    //document.getElementById(idHTML).style.backgroundColor = "white";
    document.getElementById(idHTML).style.position = "absolute";
    document.getElementById(idHTML).style.left = this.posicionCentrada.x + "px";
    document.getElementById(idHTML).style.top = this.posicionCentrada.y + "px";
    document.getElementById(idHTML).style.width = this.ancho + "px";
    document.getElementById(idHTML).style.height = this.alto + "px";
    document.getElementById(idHTML).style.zIndex = "10";
    document.getElementById(idHTML).style.background = "url('"+ this.rutaHojaSprites +"')";
    document.getElementById(idHTML).style.backgroundPosition = "-"+this.origenXSprite + "px -"+ this.origenYSprite + "px";
    document.getElementById(idHTML).style.backgroundClip = "border-box";
    document.getElementById(idHTML).style.outline = "1px solid transparent";
}
```

Por último, incluimos las líneas necesarias con estilo CSS para que, por fin, podamos ver a nuestro personaje en el mapamundi. Necesitaremos la url de la imagen, que la tenemos guardada en una variable, tambien necesitamos el **`backgroundPosition`** ya que esto nos permite elegir un sector concreto de nuestra hoja de sprites. Continuamos con el **`backgroundClip`**, que lo usaremos para evitar que el fondo se nos salga de la zona que hemos delimitado y, por último, añadimos un borde falso para que no haya bordes subrayados no deseados de HTML.

Dirigir al personaje

```
JugadorMapamundi.prototype.dirigir = function()
{
    if(this.estadoJuego == listadoEstados.MAPAMUNDI)
    {
        if(this.velocidadY > 0) //arriba
        {
            this.origenXSprite = this.ancho * 3;
        }
        if(this.velocidadY < 0) //abajo
        {
            this.origenXSprite = this.ancho * 6;
        }
    }

    if(this.velocidadX > 0) //derecha
    {
        this.origenXSprite = this.ancho * 9;
    }
    if(this.velocidadX < 0) //izquierda
    {
        this.origenXSprite = this.ancho * 12;
    }
}
```

Vamos a hacer que nuestro personaje mire en el sentido al que se dirige. Para ello crearemos una función **`dirigir()`**. Analizaremos la velocidad y esta nos indicará en qué dirección no estamos moviendo. Si la velocidad de X es menor que 0, por ejemplo, nos hará ver que nos estamos dirigiendo hacia la izquierda.

Mas adelante, cuando empecemos a trabajar con los niveles, explicaremos el porqué de la línea superior del código, así que por ahora la ignoraremos.

Continuando con la explicación, podemos observar cómo existe un valor que es multiplicado por el ancho de la imagen. Su función es indicarnos en qué posición se encuentra el Sprite que queremos mostrar. Si volvemos a coger como ejemplo la dirección izquierda, vemos que el ancho está multiplicado por 12 ya que el Sprite que usaremos se encuentra en esa posición del array.

Animaciones

Crearemos la función **`animar()`** en el fichero **`JugadorMapamundi.js`**. Primero comprobaremos si nuestro personaje está quieto, si esto es así ponemos los **`framesAnimacion = 0`**. Si supera esta condición, aumentaremos los frames de animación. Como nuestro Sprite es sencillo, y solo tiene 2 pasos por cada dirección, haremos lo siguiente:

```
let paso1 = 20;
let paso2 = 40;
let origenXSpriteTemporal = this.origenXSprite;

if(this.framesAnimacion > 0 && this.framesAnimacion < paso1)
{
    origenXSpriteTemporal += this.ancho;
}
if(this.framesAnimacion >= paso1 && this.framesAnimacion < paso2)
{
    origenXSpriteTemporal += this.ancho * 2;
}
if (this.framesAnimacion == paso2)
{
    this.framesAnimacion = 0;
}
```

Crearemos dos variables temporales que solo usaremos dentro de esta función para cada paso. Cogeremos el valor de origen X del Sprite y calculamos qué animación debería ejecutarse. Para ello pondremos las condiciones aquí mostradas. En resumen, en esencia lo que iremos haciendo es ir sumando 48 pixeles por cada animación necesaria. Para ir pasando de una a otra, simplemente tendremos que ver si los frames de animación son mayores o iguales al paso 1 y menores que el 2, solo tendremos que sumarle el doble del ancho, y el resultado sería una oscilación entre el paso número 1 y el número 2. Para terminar, tenemos que indicar que si los frames de animación son 20, debemos de resetear el valor para que esté la animación siempre alternándose entre el primer paso y el segundo.

Popups

Lo que queremos es detectar la existencia de los distintos niveles para querer entrar en ellos. Vamos a necesitar un par de clases nuevas de ayuda. Una de ellas se va a encargar de guardar las localizaciones de los distintos niveles. Crearemos **`Localizacion.js`** que guardará un rectángulo y un nombre, por ahora.

A continuación, necesitaremos otro archivo que se encargará de darnos los mensajes cuando atravesemos una posición donde podamos entrar, es decir, que abra un diálogo; **`popup.js`**.

Comencemos creando una propiedad que nos indica si el cuadro de texto está visible o no, otra función básica llamada **`mostrar`** y una ultima llamada **`ocultar`**. Para mostrar el mensaje pediremos una coordenada (x, y), un ancho y el texto. Si el mensaje ya estuviera visible, no intentaríamos mostrarlo de nuevo, así que usaremos un **`return`**. Convertimos las coordenadas a números enteros, porque cabe la posibilidad que vengan en decimales, así que usaremos la propiedad **`Math.floor()`** para X e Y. También le daremos una variable temporal "**popup**" y escribiremos las reglas CSS necesarias para mostrar el diálogo. Por ultimo, haremos que el **`popup`** sea visible.

```
mostrar: function(x, y, ancho, texto) {
    if(popup.visible) {
        return;
    }

    x = Math.floor(x);
    y = Math.floor(y);

    let id = "popup";

    document.getElementById(id).innerHTML = texto;
    document.getElementById(id).style.display = "block";
    document.getElementById(id).style.position = "absolute";
    document.getElementById(id).style.transform = 'translate3d('+ x + 'px, ' + y + 'px, 0' + ')';
    document.getElementById(id).style.width = ancho + "px";
    document.getElementById(id).style.zIndex = "11";
    document.getElementById(id).style.backgroundColor = "black";
    document.getElementById(id).style.color = "white";
    document.getElementById(id).style.border = "3px solid white";
    document.getElementById(id).style.padding = "0.5em";
    document.getElementById(id).style.textAlign = "center";

    popup.visible = true;
},
```

Ahora haremos que nuestro popup se oculte, lo cual es mucho más simple. Si no se está mostrando no intentaremos ocultarlo, le creamos una id, y le indicaremos que desaparezca del flujo de la página con un **`.display = "none"`**, y un **`innerHTML`** esté vacío. Por último, pondremos **`.visible`** en falso y ya estaría.

```
ocultar: function() {
    if(!popup.visible) {
        return;
    }

    let id = "popup";

    document.getElementById(id).style.display = "none";
    document.getElementById(id).innerHTML = "";

    popup.visible = false;
}
```

Ahora, para hacer que funcione, debemos programar la lógica en el archivo **`EstadoMapamundi`**. Debemos detectar cuándo el jugador se ha detectado con una localización, así que para ello

debemos asegurarnos que estamos creando los rectángulos para las localizaciones en el fichero **Mapa.js** de la siguiente manera:

```
if (datosCapas[i].name == "localizaciones")
{
    for (l = 0; l < datosCapas[i].objects.length; l++)
    {
        this.rectangulosLocalizaciones.push(new Localizacion(new Rectangulo(
            datosCapas[i].objects[l].x, datosCapas[i].objects[l].y,
            datosCapas[i].objects[l].width, datosCapas[i].objects[l].height, "localizacion"
        ), datosCapas[i].objects[l].name));
    }
}
```

Para que por fin podamos mostrar el popup de cada nivel, iremos a **EstadoMapamundi.js**, a la función **actualizar.**

```
let localizacionAtravesada = false;

for(var i = 0; i < this.mapa.rectangulosLocalizaciones.length; i++)
{
    let rActual = this.mapa.rectangulosLocalizaciones[i].rectangulo;
    let nombre = this.mapa.rectangulosLocalizaciones[i].nombre;
    let rTemporal = new Rectangulo(rActual.x + this.mapa.posicion.x,
    rActual.y + this.mapa.posicion.y, rActual.ancho, rActual.alto);
    let objetoEntradaLocalizacion = null;
    if(rTemporal.cruza(this.jugadorMapamundi.rectanguloGeneral))
    {
        localizacionAtravesada = true;
        objetoEntradaLocalizacion = registroLocalizaciones.obtenerLocalizacion(nombre);
        if(!popup.visible) {
            popup.mostrar(dimensiones.ancho / 2 - 150, dimensiones.alto / 2 - 100,
            300, nombre);
        }
        if(teclado.teclaPulsada(controlesTeclado.entrarLocalizacion))
        {
            maquinaEstados.cambiarEstado(listadoEstados.NIVEL, objetoEntradaLocalizacion);
            console.log(objetoEntradaLocalizacion);
        }
    }

    if(!localizacionAtravesada && popup.visible)
    {
        popup.ocultar();
    }
}
```

Recorremos el array, guardamos el rectángulo actual en una variable, su nombre y creamos un rectángulo temporal para calcular la coordenada dentro del juego ya que esta irá variando dependiendo de la ubicación del jugador porque el mapa se irá moviendo, y con ella, las localizaciones. Detectaremos si nos estamos cruzando con una localización, y empezaremos a controlar el popup. Si el popup no está visible, lo mostraremos e intentaremos que esté en el centro de la pantalla más o menos, pero lo moveremos un poco arriba y a la izquierda para que aparezca encima de nuestro jugador, y por último le daremos como texto el nombre.

Para terminar, controlaremos si no hemos atravesado ninguna localización, y el popup se está mostrando, lo que haremos es ocultarlo, es decir, ya no estamos en ninguna localización y no tiene sentido mostrarlo.

Listado de niveles

Crearemos un nuevo archivo llamado **RegitsroLocalizacionEntrada.js** donde crearemos un objeto local instanciable y le daremos los atributos básicos nombre, ruta del mapa, ruta de la imagen y la coordenada X inicial y coordenada la Y inicial. Iniciamos los objetos, como siempre. A continuación, debemos crear **registroLocalizaciones.js**. Este devolverá las localizaciones. Solo tendrá una función, que será **obtenerLocalizacion** que, como su propio nombre indica, nos devolverá la localización. Creamos un array y le hacemos un **push** al objeto registro de localizaciones de entrada, y le añadimos los parámetros que necesita.

```
var registroLocalizaciones =
{
    obtenerLocalizacion: function(nombreLocalizacion)
    {
        let localizaciones = new Array();
        localizaciones.push(new RegistroLocalizacionEntrada("Ciudad del Arbol Milenario", "niveles/villa48.json", "img/villa48_nivel.png", 0, 630));
        for(var i = 0; i < localizaciones.length; i++)
        {
            if(nombreLocalizacion == localizaciones[i].nombre)
            {
                return localizaciones[i];
            }
        }
    }
};
```

Para devolverlo, hacemos un bucle de las localizaciones, y si el nombre de la localización que hemos recibido coincide con el nombre de la localización del array, la devolveremos, así podemos cambiar de estado y pasar del mapamundi al nivel.

```
if(teclado.teclaPulsada(controlesTeclado.entrarLocalizacion))
{
    maquinaEstados.cambiarEstado(listadoEstados.NIVEL, objetoEntradaLocalizacion);
    console.log(objetoEntradaLocalizacion);
}
```

Para poder acceder a nuestro nivel, ahora tenemos que indicarle qué tecla será la que habrá que pulsar para poder acceder. Para ello, vamos a **controlesTeclado** le indicamos que tecla queremos. Una vez hecho esto, nos dirigimos a **EstadoMapamundi.js** y, en la función **actualizar** le decimos que si tiene la tecla pulsada de **entrarLocalización** queremos que la máquina de estados cambie al estado **NIVEL**.

Pantalla de título

Ahora queremos que, al iniciar nuestro juego, aparezca una pantalla de título con una pequeña animación, y cuando le hagamos click a la pantalla, se muestre el mapamundi y podamos empezar a jugar.

Crearemos un nuevo archivo llamado **EstadoPantallaTitulo.js** y este estado se ocupará de mover la pantalla de título. Crearemos una función con varios parámetros, como la ruta de la imagen, el id del HTML, ancho y alto de la imagen, frames de la animación y el movimiento en el eje Y. Le daremos a continuación los estilos que necesita para que aparezca en pantalla; posición absoluta, las dimensiones, borde, fondo negro, etc.

```
document.getElementById(this.idHTML).style.position = "absolute";
document.getElementById(this.idHTML).style.width = this.anchoImagen + "px";
document.getElementById(this.idHTML).style.height = this.altoImagen + "px";
document.getElementById(this.idHTML).style.background = "url('" + this.rutaImagenTitulo + "')";
document.getElementById(this.idHTML).style.backgroundClip = "border-box";
document.getElementById(this.idHTML).style.outline = "1px solid transparent";
document.getElementById(this.idHTML).style.transform = 'translate3d(' +
(dimensiones.ancho / 2 - this.anchoImagen / 2) + 'px, ' + (dimensiones.alto / 2 - this.altoImagen / 2) + 'px, 0';

document.getElementsByTagName("body")[0].style.overflow = "hidden";
document.getElementsByTagName("body")[0].style.backgroundColor = "black";
```

Ahora crearemos una función la cual nos permita mostrar el juego tras mostrar la pantalla de titulo tras hacer click en la misma cambiando el estado.

Recuperamos el **body**, accedemos al evento **onclick** y le asignamos una función anónima. Devolvemos la referencia al elemento por su id **pantalla-titulo** y que su estile esté en **none**, y hacemos que el **body** desaparezca. En esencia, conseguimos que al hacer click, el **div** de la pantalla de título desaparezca y desactivamos el click. Por último, hacemos el cambio de estado a mapamundi.

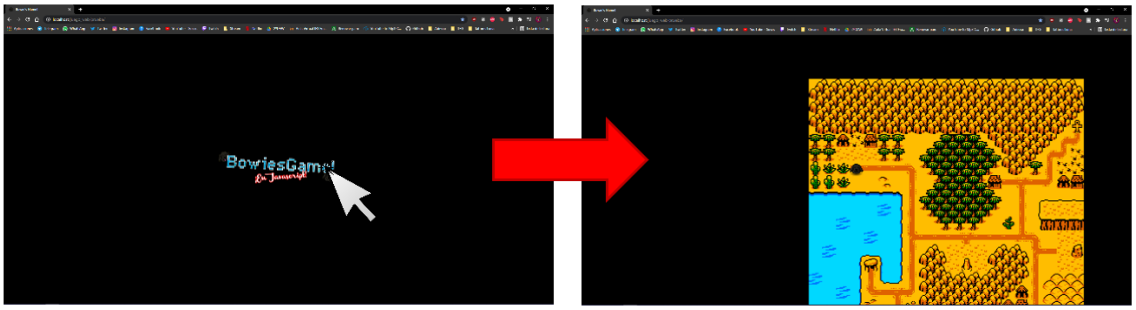
```
EstadoPantallaTitulo.prototype.actualizar = function(registroTemporal) {
    if(this.framesAnimacion < 30) {
        this.movimientoY++;
    }
    if(this.framesAnimacion >= 30 && this.framesAnimacion < 90) {
        this.movimientoY--;
    }
    if(this.framesAnimacion >= 90 && this.framesAnimacion < 120) {
        this.movimientoY++;
    }

    this.framesAnimacion++;

    if(this.framesAnimacion >=120) {
        this.framesAnimacion = 0;
        this.movimientoY = 0;
    }
}
```

Creamos una función **actualizar**, que lo usaremos para ir actualizando el movimiento. Como actualizamos 60 frames por segundo, queremos que le movimiento lo haga cada medio segundo. Con la primera condición, la imagen subirá, con la segunda bajará, y la tercera hará que vuelva a su estado original. Y cada vez que pasamos esta batería de actualizaciones, aumentamos los frames de animación en 1. Por último, hay que reiniciar el mecanismo. Si pasamos del máximo, reiniciamos todo, tanto los frames de animación, como el movimiento en el eje Y.

Lo último que nos quedaría hacer es que todo esto se dibuje y funcione.

```
EstadoPantallaTitulo.prototype.dibujar = function() {
    document.getElementById(this.idHTML).style.transform = 'translate3d(' +
        (dimensiones.ancho / 2 - this.anchoImagen / 2) + 'px, ' +
        (dimensiones.alto / 2 - this.altoImagen / 2 + this.movimientoY) + 'px, 0)';
}
```

Ahora debemos irnos a la **maquinaEstados.js** y creamos un nuevo listado de estados para la pantalla de título, además de que en la función iniciar, debemos poner nuestro nuevo estado **PANTALLA_TITULO**, para que sea el primero que se ejecute. También debemos añadirlo al **listadoEstado.js**. Una vez realizado todo este proceso, podemos recargar nuestro juego y ya debe aparecer.



Preparación de los niveles

Comenzaremos a preparar los controles y la implementación del jugador para el mapa en 2D de plataformas, así que ahora toca modificar un poco de código.

Primero nos iremos a **maquinaEstados.js** y crearemos un nuevo el atributo para **cambiarEstado** llamado **objetoEntradaLocalizacion**. Una vez hecho esto, nos vamos a **EstadoMapamundi** y añadiremos **let objetoEntradaLocalizacion**, en la función actualizar dentro del bucle que recorre los rectángulos de las localizaciones. Rellenamos el objeto para que pueda obtener el nombre de la localización t creamos una condición la cual haga que, cuando se pulsa la tecla para entrar, o salir de un nivel, esto haga que se cambie el estado del juego. Volvemos a la máquina de estados y tenemos que añadir lo siguiente.

```
if(teclado.teclaPulsada(controlesTeclado.entrarLocalizacion))
{
    maquinaEstados.cambiarEstado(listadoEstados.NIVEL, objetoEntradaLocalizacion);
}
if(teclado.teclaPulsada(controlesTeclado.salirLocalizacion))
{
    maquinaEstados.cambiarEstado(listadoEstados.MAPAMUNDI, objetoEntradaLocalizacion);
}
```

```
case listadoEstados.MAPAMUNDI:
    maquinaEstados.estadoActual = new EstadoMapamundi(listadoEstados.MAPAMUNDI, "mapas/desierto48.json", 500, 500);
    break;
case listadoEstados.NIVEL:
    maquinaEstados.estadoActual = new EstadoMapamundi(listadoEstados.NIVEL, objetoEntradaLocalizacion.rutaMapa,
        objetoEntradaLocalizacion.coordenadaXInicial, objetoEntradaLocalizacion.coordenadaYInicial);
    break;
```

En el **MAPAMUNDI** tenemos que añadir el estado en el que se encuentra, la y la posición en la que queremos que aparezca nuestro personaje, o sea, la coordenada. Para el **NIVEL** el estado, la ruta del mapa y las coordenadas del personaje.

Con esto que hemos implementado, ya se debería ver a nuestro personaje en la pantalla, pero con la peculiaridad de que podríamos movernos por toda la pantalla, sin importar que realmente es un mapa de Scroll lateral, ya que los controles no están acondicionados para este tipo de nivel.

Controles de salto

En este apartado nos centraremos en adaptar el movimiento del jugador para que, cuando entremos en el nivel del juego, podamos dirigirnos de derecha a izquierda y controlar el salto de nuestro personaje. Para ello nos dirigimos al archivo **Mapa.js** y añadimos un estado de juego al mapa. Como iremos cambiando entre mapas y niveles, debemos de crear una condición para alternar entre ellos para poder seleccionar una imagen u otra dependiendo del estado del juego.

```
if (this.estadoJuego == listadoEstados.MAPAMUNDI)
{
    this.rutaImagenMapa = "img/" + nombreMapa[0] + ".mapa.png";
}
if (this.estadoJuego == listadoEstados.NIVEL)
{
    this.rutaImagenMapa = "img/" + nombreMapa[0] + ".nivel.png";
}
```

A continuación, nos dirigimos a **`JugadorMapamundi.js`** y en su propia función, le añadimos el atributo **`estadoJuego`**, porque, al igual que pasa con el mapa, cuando cambiemos de mapamundi a nivel, también debemos cambiar el tipo de control del personaje.

Comenzamos creando mas controles, porque hasta ahora solo controlábamos la velocidad X y la Y, ya que nos permitían movernos en las 4 direcciones sin problema, pero, al movernos ahora de lado, debemos "eliminar una dimensión", que sería movernos arriba y abajo. Sin embargo, añadimos la complicación de que debemos controlar un salto vertical y caernos.

Debemos añadir lo siguiente a nuestro código:

```
this.subiendo = false;
this.saltoBloqueado = false;
this.saltoYInicial = 0;
this.framesAereosMaximos = 12;
this.framesAereos = this.framesAereosMaximos;

this.velocidadTerminal = 10;
this.velocidadCaida = 0;
```

Subiendo	Indica si cuando estamos saltando, si estamos ganando altura o no
Salto bloqueado	Bloqueamos la posibilidad de hacer varios saltos entando en el aire. No poder saltar hasta tocar el suelo
Salto Y inicial	Mide desde qué altura empezamos a saltar
Frames aéreos máx.	Cuantos frames podemos pasar en el aire
Frames aéreos	Contador de cuantos frames nos quedan en el aire
Velocidad terminal	Velocidad máxima a la que caerá
Velocidad caída	Velocidad que modificaremos para que el personaje acelere o frene

Ahora debemos separar los controles, uno para mover al personaje en el mapamundi, y otro para moverlo en el nivel. En esencia **`moverEnMapamundi`** se quedará tal y como estaba originalmente, ya que los controles están perfectamente preparados para ese uso, sin embargo, para **`moverEnNivel`** debemos hacer ciertas modificaciones.

Realmente todo funciona igual, con la única diferencia de que crearemos un sistema de gravedad, es decir, siempre estaremos cayendo hacia abajo para estar pegados a la colisión del suelo. Escribiremos las siguientes condiciones:

- Si el salto está bloqueado, si nos estamos chocando hacia abajo y si no hemos pulsado la tecla de saltar, lo que haremos será movernos sobre el mapa.
- Si no tenemos el salto bloqueado y no estamos pulsando la tecla de saltar tenemos la opción de saltar y automáticamente, lo bloquearemos para no poder volver a saltar.
- Si no tenemos una colisión arriba y estamos subiendo, lo que haremos será reducir los frames aéreos y la velocidad del eje Y será igual a 1 multiplicado por la velocidad de movimiento más los frames aéreos. Lo que conseguimos con esto es poder subir en el salto rápidamente, pero a medida que va llegando a su altura máxima, la velocidad incrementará más lentamente hasta llegar al punto de parar y volver a caer hacia abajo. Para esto último, debemos de comprobar si los frames aéreo son menores o iguales a 0
- Si los frames aéreos son menores o iguales a 0, os sea, no podemos subir más, tendremos que indicar que ya no estamos subiendo y que los frames aéreos han llegado a su máximo, o también se puede decir que reiniciamos el contador.

- Si no estamos colisionando con nada abajo, y no estamos subiendo, o sea, estamos en el aire (cayendo), tenemos que hacer que la velocidad Y se acelere mientras caemos sin que haya decimales, para que sea constante. Si la velocidad de la caída es menor que la velocidad terminal, le iremos sumando 0.3 a la caída, que es un valor que hace que la caída se vea lo más natural posible.
- Para movernos a los lados comprobamos si no tenemos una colisión a la izquierda, por ejemplo, y estemos pulsando la tecla izquierda hacemos que la velocidad X sea 1 * la velocidad de movimiento, y lo mismo para la derecha.

Ahora debemos poder movernos a izquierda y derecha, saltar y caer cuando entremos en un nivel, pero si nos mantenernos en el mapamundi, nada de esto se aplica, y nos podremos mover en cualquier dirección.

Animaciones laterales

Actualmente, la animación de saltar no funciona correctamente, ya que el juego interpreta que, al saltar, nos dirigimos hacia arriba, y cuando bajamos, miramos hacia abajo. Lo que nosotros queremos es que nuestro personaje siga mirando hacia la misma dirección, pero en el salto, tanto en la subida como en la bajada.

```
JugadorMapamundi.prototype.dirigir = function()
{
    if(this.estadoJuego == listadoEstados.MAPAMUNDI)
    {
        if(this.velocidadY > 0) //arriba
        {
            this.origenXSprite = this.ancho * 3;
        }
        if(this.velocidadY < 0) //abajo
        {
            this.origenXSprite = this.ancho * 6;
        }
    }

    if(this.velocidadX > 0) //derecha
    {
        this.origenXSprite = this.ancho * 9;
    }
    if(this.velocidadX < 0) //izquierda
    {
        this.origenXSprite = this.ancho * 12;
    }
}
```

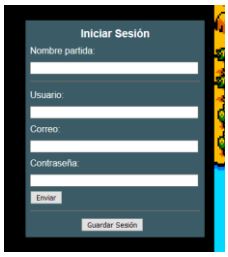
Primero, adaptaremos nuestro **`dirigir`** para que compruebe el estado del juego, si está en un nivel o en el mapamundi. Gracias a esta condición, lo que conseguimos es bloquear el movimiento vertical en los niveles de plataformas, y en el mapamundi, no.

Lo que haremos será modificar un poco la función **`animar`** de la siguiente manera:

```
JugadorMapamundi.prototype.animar = function() {
    if(this.velocidadX == 0 && this.velocidadY == 0) {
        this.framesAnimacion = 0;
        return;
    }

    this.framesAnimacion++;

    let paso1 = 20;
    let paso2 = 40;
    let origenXSpriteTemporal = this.origenXSprite;

    if(this.framesAnimacion > 0 && this.framesAnimacion < paso1) {
        origenXSpriteTemporal += this.ancho;
    }
    if(this.framesAnimacion >= paso1 && this.framesAnimacion < paso2) {
        origenXSpriteTemporal += this.ancho * 2;
    }
    if(this.framesAnimacion == paso2) {
        this.framesAnimacion = 0;
    }

    document.getElementById("jugador").style.backgroundPosition = "-" + origenXSpriteTemporal + "px -" + this.origenYSprite + "px";
}
```

Si no nos movemos, el estado de la animación vuelve a 0, si no podemos seguir saltando, bloquearemos las siguientes partes de la animación, porque queremos que la animación se mantenga hasta que vuelva a tocar el suelo. El resto del código se quedaría prácticamente igual.

Ventana de inicio de sesión



Invocaremos un rectángulo con unos colores y diseño totalmente personalizable donde podremos encontrar las opciones de iniciar sesión y guardar partida.

Comenzamos creando el div y las etiquetas HTML necesarias para introducir todos los datos en el fichero **index**. Copiamos y pegamos la clase **popup** y la pegamos en un nuevo fichero llamado **inicioSesion** ya que tenemos la suerte de poder reciclar gran parte del código.

Comenzamos reemplazando **popup** > **inicioSesion**. Añadimos **ultimoRegistro: 0**, el uso que tendrá será para saber cuando se abrió para evitar ciertas complicaciones a la hora de abrir y cerrar la ventana. Sustituimos en la función mostrar el ancho y el texto por un registro temporal que lo alimentamos en el bucle principal cada vez que actualizamos y dibujamos el juego. Fijamos el ancho, creamos una id para aplicarle los estilos a la ventana, y aplicamos los estilos que deseemos.

```
document.getElementById(id).style.border = "3px solid black";
document.getElementById(id).style.display = "block";
document.getElementById(id).style.lineHeight = "30px";
document.getElementById(id).style.position = "absolute;";
document.getElementById(id).style.transform = 'translate3d('+ x + 'px, ' + y + 'px, 0' + ')';
document.getElementById(id).style.width = ancho + "px";
document.getElementById(id).style.zIndex = "11";
document.getElementById(id).style.backgroundColor = "#3B5C66";
document.getElementById(id).style.color = "white";
document.getElementById(id).style.border = "3px solid black";
document.getElementById(id).style.padding = "0.5em";
document.getElementById(id).style.fontFamily = "sans-serif, Helvetica";
```

Seguimos creando una función llamada **listoParaCambiar** , que usará el registro temporal como parámetro. Su finalidad será comprobar saber hace cuanto tiempo pulsamos la tecla para abrir la ventana para poder aplicarle un "delay" a la hora de abrirlo. Esto se hace para que, si el usuario le hace click muchas veces seguidas o mantiene el botón pulsado, no esté abriéndose y cerrándose continuamente; tendrá un mínimo retraso de 200 milisegundos.

```
listoParaCambiar: function(registroTemporal)
{
    if(registroTemporal - inicioSesion.ultimoRegistro < 200)
    {
        return false;
    }
    else
    {
        inicioSesion.ultimoRegistro = registroTemporal;
        return true;
    }
}
```

Una vez hecho esto, vamos a la parte del código donde le indicamos que, si la ventana ya está visible, no podamos volver a abrirla, y le añadimos que, si la ventana no está lista para cambiar y lo alimentamos con el registro temporal, le hacemos un return. Así nos podremos saltar la función. Lo mismo se lo aplicamos a la función **ocultar**.

Lo que nos quedaría sería borrar la línea del **innerHTML** y dirigirnos al fichero **controlesTeclado** y asignarle una tecla la cuál tendrá la función de hace que aparezca y desaparezca nuestra ventana de inicio de sesión[ç].

Para finalizar, nos dirigimos a E**stadoMapamundi** hay que añadir un poco de código justo debajo de donde aplicamos el popup, incluimos lo siguiente:

```
if(teclado.teclaPulsada(controlesTeclado.inicioSesion) && !inicioSesion.visible)
{
    inicioSesion.mostrar(100,100, registroTemporal);
}
if(teclado.teclaPulsada(controlesTeclado.inicioSesion) && inicioSesion.visible)
{
    inicioSesion.ocultar(registroTemporal);
}
```

Es simple, lo que hará es que aparezca y desaparezca nuestra ventana de inicio de sesión usando la función **mostrar** y **ocultar**. En el **mostrar** es necesario también una X y una Y, la cual añadimos justo antes del registro temporal.