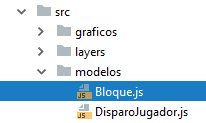
|  |
| --- |
| Software de entretenimiento y Videojuegos Jordán Pascual : pascualjordan@uniovi.es |

|  |  |
| --- | --- |
| 3 | Plataformas - parte 2 |

# Bloques

Vamos a agregar un nuevo modelo **Bloque**, este elemento nos servirá para construir: suelos, muros, incluso fondos.

Creamos el fichero **src/modelos/Bloque.js** .



|  |
| --- |
| **class** Bloque **extends** Modelo {   constructor(rutaImagen, x, y) {  **super**(rutaImagen, x, y)  }  } |

Agregamos el script **Bloque.js** al **index.html** (Debe incluirse después de **modelo**)

|  |
| --- |
| <**script src="src/modelos/Modelo.js"**></**script**> <**script src="src/modelos/Bloque.js"**></**script**> <**script src="src/modelos/DisparoJugador.js"**></**script**> |

Abrimos el **GameLayer.js**  y en su método **iniciar()** creamos un array de **bloques** vacío.

|  |
| --- |
| iniciar() {  *reproducirMusica*();  **this**.**bloques** = [];  **this**.**fondoPuntos** =  **new** Fondo(***imagenes***.**icono\_puntos**, 480\*0.85,320\*0.05); |

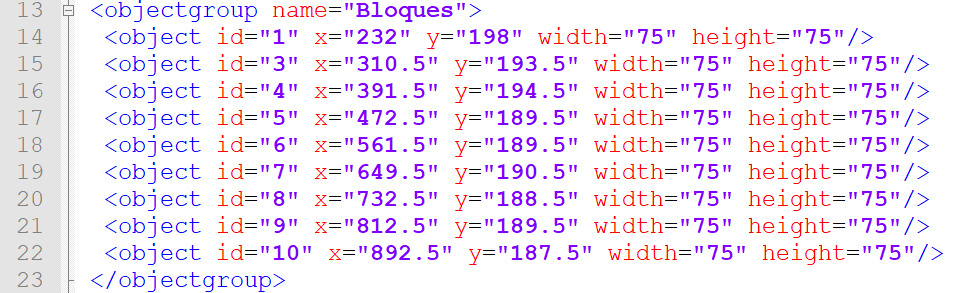
Recordamos que como el resto de modelos, también debemos **dibujar()** los bloques, lo hacemos justo después del fondo.

|  |
| --- |
| dibujar (){  **this**.**fondo**.dibujar();  **for** (**var** i=0; i < **this**.**bloques**.length; i++){  **this**.**bloques**[i].dibujar();  } |

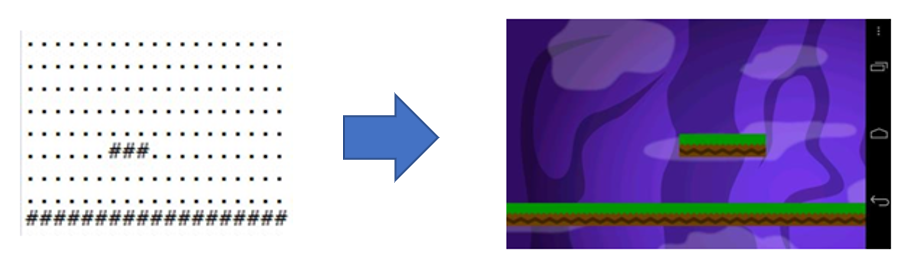
# Definición de mapas

Los niveles de los juegos suelen ser serializados en ficheros. El fichero indica donde colocar cada uno de los elementos del juego (jugador, enemigos, monedas, etc.)

Técnicamente podemos usar cualquier formato para serializar los elementos de un nivel:



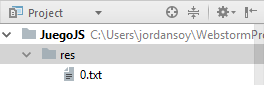
Ejemplo nivel formalizado en xml (formato tmx)



Hay algunos formatos “estándar” como el tmx (lo veremos en el bloque 2).

En esta práctica vamos a usar una matriz (muy similar a las que usan muchos juegos basados en mapas de tiles estrictos, aunque nuestro juego no va a ser estricto).

La ventaja de los mapas en forma de matriz que se asemejan mucho al nivel y son fáciles de “interpretar” y modificar sin ningún programa adicional.  
  
Abrimos el fichero **/res/0.txt** este será el mapa del nivel 0. El mapa más simple de todos.



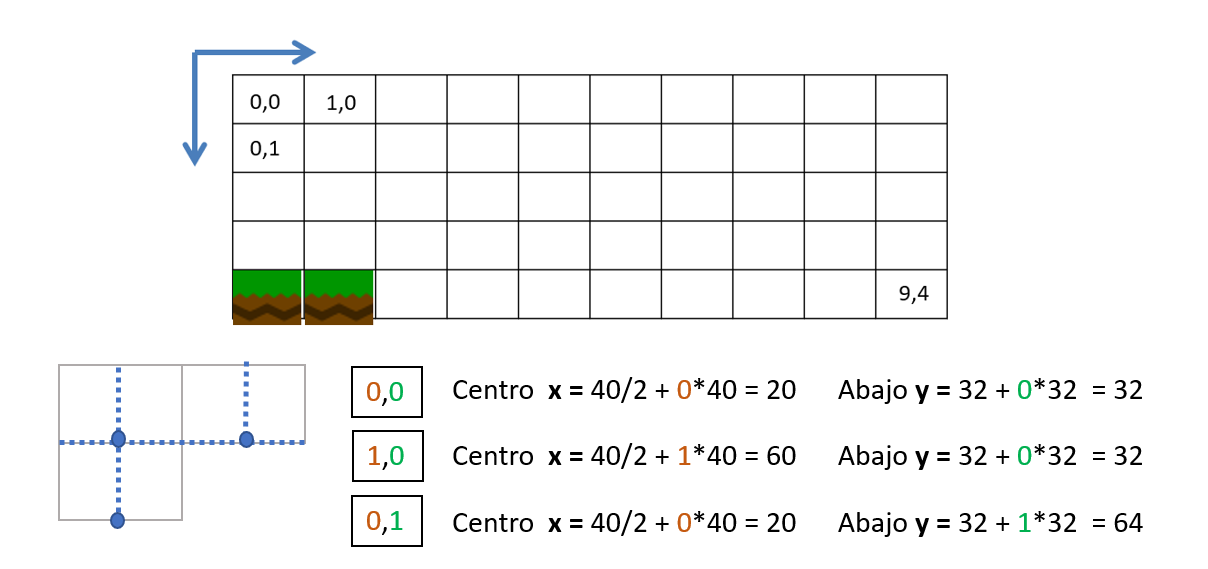
|  |
| --- |
| ............................................. ............................................. ............................................. ............................................. ..####................#...................... ..........####.....................###.#..... ............................................. ..1...................#....................## #######..#################################### |

Modificamos el **GameLayer.js,** enprimer lugar eliminamos la creación del **jugador** y **enemigos** e incluimos una llamada al método **cargarMapa(src mapa)** aun sin implementar

|  |
| --- |
| iniciar() {  *reproducirMusica*();  **this**.**bloques** = [];  **this**.**fondoPuntos** =  **new** Fondo(***imagenes***.**icono\_puntos**, 480\*0.85,320\*0.05);   **this**.**puntos** = **new** Texto(0,480\*0.9,320\*0.07 );   **~~this~~**~~.~~**~~jugador~~** ~~=~~ **~~new~~** ~~Jugador(50, 50);~~  **this**.**fondo** = **new** Fondo(***imagenes***.**fondo\_2**,480\*0.5,320\*0.5);   **this**.**disparosJugador** = []  **this**.**enemigos** = [];  **~~this~~**~~.~~**~~enemigos~~**~~.push(~~**~~new~~** ~~Enemigo(300,50));~~ **~~this~~**~~.~~**~~enemigos~~**~~.push(~~**~~new~~** ~~Enemigo(350,200));~~   **this**.cargarMapa(**"res/0.txt"**);  } |

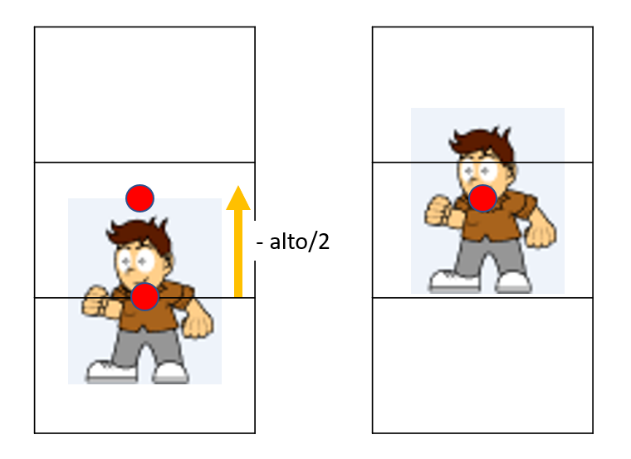
Implementamos **cargarMapa(),** se obtiene el fichero **0.txt** realizando una petición HTTP GET, luego se lee ese fichero línea a línea, y dentro de cada línea símbolo a símbolo.

Suponemos que cada celda de la matriz tiene un tamaño de 40x32 (un tamaño comúnmente utilizado para los bloques/tiles). Solo debemos ir recorriendo celda a celda y calcular el punto que hace de suelo de cada celda (es más fácil posicionar cosas encima de un tile usando el suelo como referencia en lugar del punto central, sobre todo si los elementos no miden lo mismo que los tiles ¿Qué tile es el centro de un elemento que mide 45? )



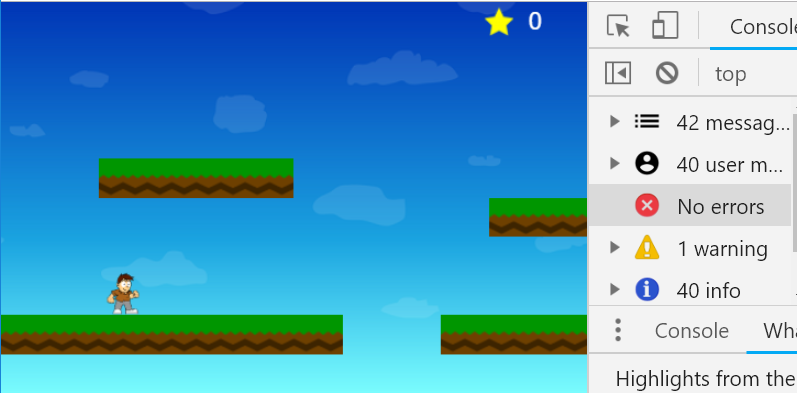
|  |
| --- |
| cargarMapa(ruta){  **var** fichero = **new *XMLHttpRequest***();  fichero.open(**"GET"**, ruta, **false**);    fichero.onreadystatechange = **function** () {  **var** texto = fichero.responseText;  **var** lineas = texto.split(**'\n'**);  **for** (**var** i = 0; i < lineas.length; i++){  **var** linea = lineas[i];  **for** (**var** j = 0; j < linea.length; j++){  **var** simbolo = linea[j];  **var** x = 40/2 + j \* 40; *// x central* **var** y = 32 + i \* 32; *// y de abajo* **this**.cargarObjetoMapa(simbolo,x,y);  }  }  }.bind(**this**);   fichero.send(**null**); } |

Implementamos el método **cargarObjetosMapa(símbolo,x y).** Como sabemos que la **x** e **y** representan el suelo pero **nuestros modelos** usan la x y la centrales, tenemos que “moverlos hacia arriba”, restándoles la mitad de su alto. (es igual para todos los elementos)

****

|  |
| --- |
| cargarObjetoMapa(simbolo, x, y){  **switch**(simbolo) {  **case "1"**:  **this**.**jugador** = **new** Jugador(x, y);  *// modificación para empezar a contar desde el suelo* **this**.**jugador**.**y** = **this**.**jugador**.**y** - **this**.**jugador**.**alto**/2;  **break**;  **case "#"**:  **var** bloque = **new** Bloque(***imagenes***.**bloque\_tierra**, x,y);  bloque.**y** = bloque.**y** - bloque.**alto**/2;  *// modificación para empezar a contar desde el suelo* **this**.**bloques**.push(bloque);  **break**;  } } |

Ejecutamos el juego y comprobamos que no tiene errores y que tanto los **bloques** como el **jugador** aparecen en pantalla.



# Cargar enemigos en el mapa

En esta nueva versión del juego dejaremos de generar enemigos de forma automática. Eliminamos la generación de enemigos del método **actualizar()** del **GameLayer**.

|  |
| --- |
| actualizar (){  **this**.**fondo**.**vx** = -1;  **this**.**fondo**.actualizar();  *~~// Generar Enemigos~~* **~~if~~** ~~(~~**~~this~~**~~.~~**~~iteracionesCrearEnemigos~~** ~~==~~ **~~null~~**~~){~~ **~~this~~**~~.~~**~~iteracionesCrearEnemigos~~** ~~= 0;  }~~ *~~// iteracionesCrearEnemigos tiene que ser un número~~* **~~this~~**~~.~~**~~iteracionesCrearEnemigos~~** ~~++;~~ **~~if~~** ~~(~~ **~~this~~**~~.~~**~~iteracionesCrearEnemigos~~** ~~> 110){~~ **~~var~~** ~~rX =~~ ***~~Math~~***~~.random() \* (600 - 500) + 500;~~ **~~var~~** ~~rY =~~ ***~~Math~~***~~.random() \* (300 - 60) + 60;~~ **~~this~~**~~.~~**~~enemigos~~**~~.push(~~**~~new~~** ~~Enemigo(rX,rY));~~ **~~this~~**~~.~~**~~iteracionesCrearEnemigos~~** ~~= 0;  }~~   **this**.**jugador**.actualizar(); |

Modificamos el fichero **0.txt** e incluimos varios símbolos **E** donde anteriormente había puntos (los símbolos **E** van a representar a los enemigos).

|  |
| --- |
| ............................................. ............................................. ............................................. .....E....................................... ..####.....E..........#...................... ..........####.....................###.#..... ............................................. ..1.........E.........#....................## #######..#################################### |

Ampliamos la función **cargarObjetoMapa()** para que reconozca el símbolo **E** y cree un **enemigo** sobre el suelo de esa celda.

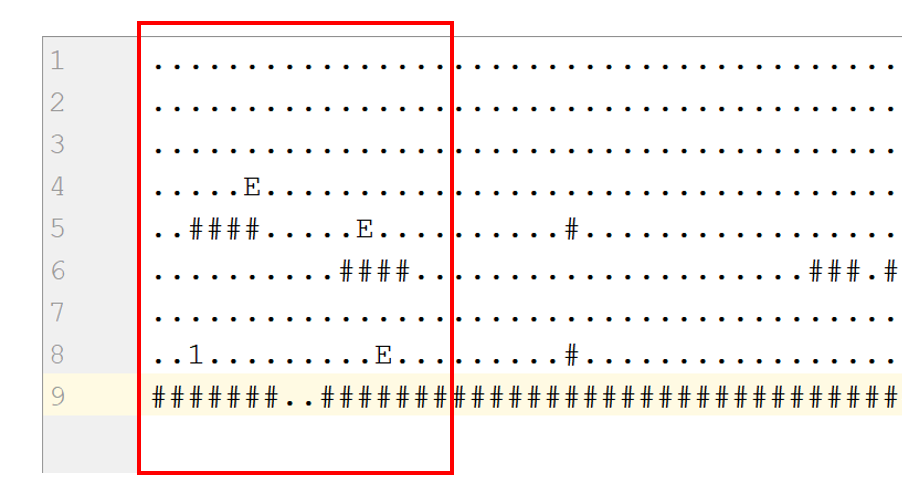
|  |
| --- |
| cargarObjetoMapa(simbolo, x, y){  **switch**(simbolo) {  **case "E"**:  **var** enemigo = **new** Enemigo(x,y);  enemigo.**y** = enemigo.**y** - enemigo.**alto**/2;  *// modificación para empezar a contar desde el suelo* **this**.**enemigos**.push(enemigo);  **break**;  **case "1"**:  **this**.**jugador** = **new** Jugador(x, y); |

Probamos que la aplicación funciona sin errores, los enemigos deben aparecen donde los hemos colocado.



# Scroll

En muchos juegos el mapa del nivel es mayor que la pantalla visible (canvas), en nuestro caso hemos dicho que la pantalla tiene 480 de ancho, como observamos eso solo nos sirve para representar una pequeña parte del mapa. **¿Cómo podemos visualizar las otras partes del mapa?**



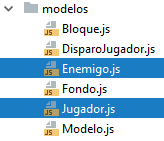
El Jugador, los bloques y enemigos pueden estar en coordenadas mucho más lejanas a las 480, lo que debemos hacer es diferenciar la coordenada X que posee un elemento de la coordenada X en la que se dibuja en el canvas.

|  |
| --- |
| **Importante:** La coordenada real donde está el Jugador (u otro elemento) en el mapa, no tiene porque ser la misma coordenada que en el Canvas.  Ej. el **Jugador** puede haber avanzado y estar en la coordenada X=3000 del mapa, pero en la pantalla/canvas se dibuja en la x=0 (en una coordenada visible dentro de 0-480) |

Comenzamos modificando el **dibujar()** de **Modelo.js**, para que no los dibuje los modelos en su coordenada real. Sí no nos envían un valor para el **scrollX** sí que los dibujaremos en su posición **x** real (Esto nos servirá para elementos que realmente NO son parte del nivel los **marcadores, fondos** y otros elementos que no son parte del mapa).

|  |
| --- |
| dibujar (scrollX){  scrollX = scrollX || 0;  ***contexto***.drawImage(**this**.**imagen**,  **this**.**x** - **this**.**imagen**.width/2 - scrollX,  **this**.**y** - **this**.**imagen**.height/2); } |

**Recordamos**, que **Enemigo.js** y **Jugador.js** tiene el método **dibujar()** **redefinido** por lo que debemos incluir el **scrollX** también en estos dos métodos.



|  |
| --- |
| dibujar (scrollX){  scrollX = scrollX || 0;  **this**.**animacion**.dibujar(**this**.**x** - scrollX, **this**.**y**); } |

Repetir en **Enemigo.js**

|  |
| --- |
| dibujar (scrollX){  scrollX = scrollX || 0;  **this**.**animacion**.dibujar(**this**.**x** - scrollX, **this**.**y**); } |

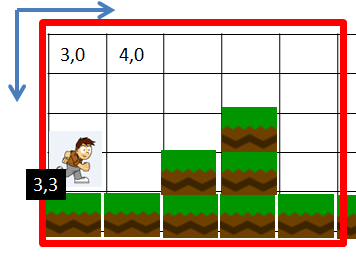
Repetir en **Jugador.js**

Accedemos al **GameLayer** y comenzamos declarando la variable **srcollX** en el constructor.

|  |
| --- |
| iniciar() {  **this**.**scrollX** = 0; |

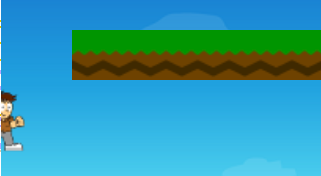
Dentro del **GameLayer** creamos el método **calcularScroll().** Para incluir un **scrollX** en el mapa basta con tomar un punto de referencia de algo que queramos que se vea siempre en pantalla (Ej: la posición del **jugador**) y restar esa medida a todos los elementos que dibujemos y que pertenezca al mapa (bloques y enemigos).

**¿Cómo hacemos que el jugador se vea si esta fuera del Canvas ej en la x=3000?, al dibujarlo le restamos 3000 jugador.x a todo así acaba en la 0 que si que está dentro del Canvas!**

****

|  |
| --- |
| calcularScroll(){  **this**.**scrollX** = **this**.**jugador**.**x**; }  dibujar (){  **this**.calcularScroll();  **this**.**fondo**.dibujar();  **for** (**var** i=0; i < **this**.**bloques**.length; i++){  **this**.**bloques**[i].dibujar(**this**.**scrollX**);  }  **for** (**var** i=0; i < **this**.**disparosJugador**.length; i++) {  **this**.**disparosJugador**[i].dibujar(**this**.**scrollX**);  }  **this**.**jugador**.dibujar(**this**.**scrollX**);  **for** (**var** i=0; i < **this**.**enemigos**.length; i++){  **this**.**enemigos**[i].dibujar(**this**.**scrollX**);  }  **this**.**fondoPuntos**.dibujar();   **this**.**puntos**.dibujar(); } |

Si probamos el juego veremos que ahora podemos movernos por el mapa y todo comienza a dibujarse a partir de **la posición X del jugador (punto central)**.



Para dar algo de margen fijo podríamos oficiar la función **calcularScroll()** añadiéndole unos 200px a la posición del jugador.

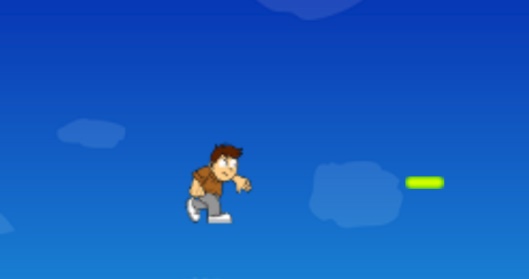
|  |
| --- |
| calcularScroll(){  **this**.**scrollX** = **this**.**jugador**.**x** - 200; } |

**Probamos la aplicación.**

# Método obsoleto Modelo.estaEnpantalla()

# Importante Modelo.estaEnpantalla() ya no funciona, Probablemente no lo hayamos percibido, pero tenemos un problema con la función Modelo.estaEnpantalla() . Anteriormente esta función detectaba si las coordenadas XY estaban dentro de los primeros 480 pixels (cuando el mapa media lo mismo que el canvas), pero ahora esta función debe ser modificada.

**Prueba:** si nos movemos más de 480px hacia delante y tratamos de disparar los disparos desaparecerán automáticamente, al tener una x > 480.



Debemos tener en cuenta el **scrollX** también en la función **estaEnPantalla()** de **Modelo.js.**

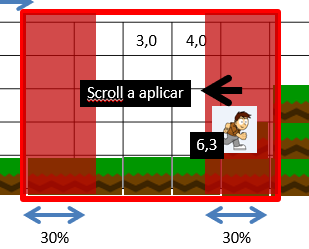
|  |
| --- |
| estaEnPantalla (){  **if** ( (**this**.**x** - ***gameLayer***.scrollX) - **this**.**ancho**/2 <= 480 &&  (**this**.**x** - ***gameLayer***.scrollX) + **this**.**ancho**/2 >= 0 &&  **this**.**y** - **this**.**alto**/2 <= 320 &&  **this**.**y** + **this**.**alto**/2 >= 0 ){  **return true**;  }  **return false**; } |

# Scroll Variable 1 – Límites de pantalla/canvas

Algunos juegos no modifican el valor el **Scroll** cada vez que el **Jugador** se mueve un pixel, sino que establecen un límite superior e inferior, el scroll solo comienza a modificarse cuando se superan esos límites.

En el siguiente ejemplo se modifica el scroll :

1. Cuando el **Jugador** está cerca del 30% de la pantalla por la izquierda**,**
2. Cuando el **jugador** está cerca 30% del por la izquierda el (100-30 = 70%)



|  |
| --- |
| calcularScroll(){  **~~this~~**~~.~~**~~scrollX~~** ~~=~~ **~~this~~**~~.~~**~~jugador~~**~~.~~**~~x~~** ~~- 200;~~  *// limite izquierda* **if** ( **this**.**jugador**.**x** - **this**.**scrollX** < 480\*0.3){  **this**.**scrollX** = **this**.**jugador**.**x** - 480\*0.3;  }  *// limite derecha* **if** ( **this**.**jugador**.**x** - **this**.**scrollX** > 480\*0.7){  **this**.**scrollX** = **this**.**jugador**.**x** - 480\*0.7;  } } |

# Scroll Variable 2 – Limites de mapa

Además de los límites de la pantalla/canvas (480) también se pueden establecer límites con el tamaño del mapa, de esta forma evitaremos que el Scroll enfoque a partes donde no hay mapa (por el principio del mapa o por el final).

El scroll **SOLO** se aplicará si:

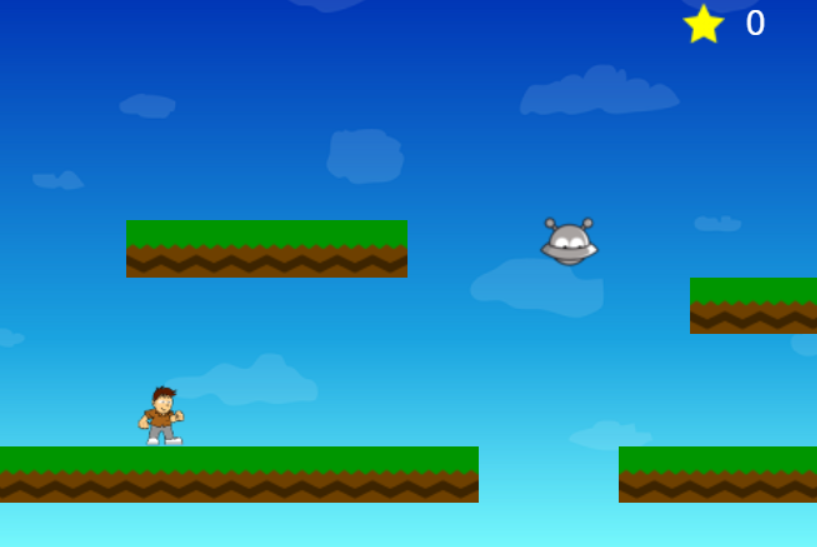
1. Cuando el **Jugador** está más lejos del 30% del inicio del mapa, más lejos de los 480\*0.3 primeros pixeles
2. Cuando el **jugador** aún no ha llegado al último 480\*0.3 del mapa, hace falta saber el tamaño del mapa.

|  |
| --- |
| calcularScroll(){  *// limite izquierda* **if** ( **this**.**jugador**.**x** > 480 \* 0.3) {  **if** (**this**.**jugador**.**x** - **this**.**scrollX** < 480 \* 0.3) {  **this**.**scrollX** = **this**.**jugador**.**x** - 480 \* 0.3;  }  }   *// limite derecha* **if** ( **this**.**jugador**.**x** < **this**.**anchoMapa** - 480 \* 0.3 ) {  **if** (**this**.**jugador**.**x** - **this**.**scrollX** > 480 \* 0.7) {  **this**.**scrollX** = **this**.**jugador**.**x** - 480 \* 0.7;  }  } } |

En el método **cargarMapa()** debemos calcular el valor de **this.anchoMapa** teniendo en cuenta el número de bloques que hay y su ancho.

|  |
| --- |
| cargarMapa(ruta){  **var** fichero = **new *XMLHttpRequest***();  fichero.open(**"GET"**, ruta, **false**);   fichero.onreadystatechange = **function** () {  **var** texto = fichero.responseText;  **var** lineas = texto.split(**'\n'**);  **this**.**anchoMapa** = (lineas[0].length-1) \* 40;  **for** (**var** i = 0; i < lineas.length; i++){  **var** linea = lineas[i]; |

# Probar el juego



# Controlar el movimiento libre - Físicas

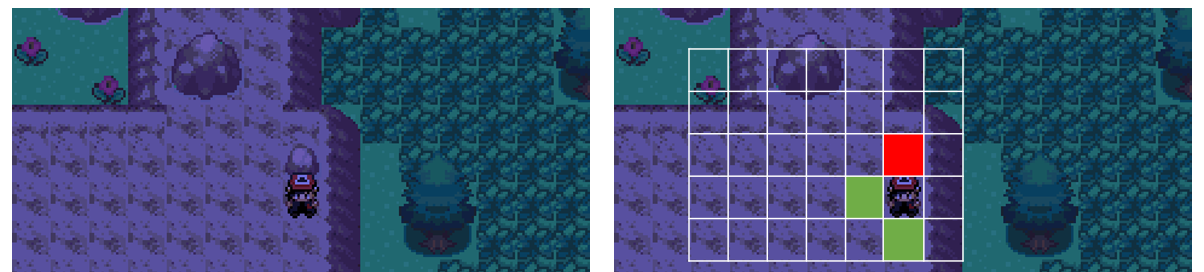
Hasta ahora en el juego existe movimiento libre, cada elemento **Bloques**, **Jugador**, **Enemigo** y **DisparoJugador**, se mueve (modifica sus coordenadas X e Y) sin necesidad de consultar al resto de los elementos del juego.

Si necesitáramos que el movimiento de los elementos se hiciese de una **forma dependiente,** necesitaríamos tener en cuenta todos los elementos del juego antes de moverlos. Esto se suele hacer utilizando un **motor de movimiento**, analiza la posición y velocidad de todos los elementos y los mueve.

Para evitar que dos elementos del juego puedan “sobreponerse” necesitaremos implementar ese **motor de movimiento**. (Realmente el motor también podría manejar elementos de distintos tipos, “**solidos**” y “**traspasables**”).

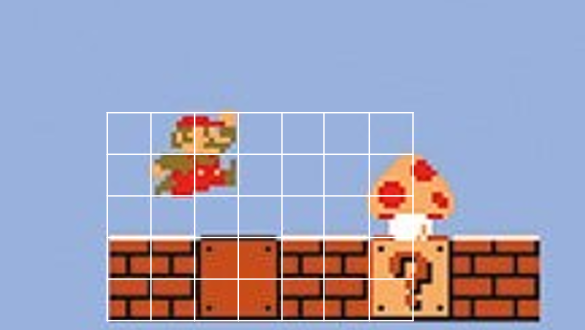
**Mapas de tile**

La forma más sencilla de controlar el movimiento es mediante un mapa de **tiles estricto**, consiste básicamente en que cada elemento ocupa una posición de la matriz. Se mueven de celda en celda, comprobando si la celda a la que quiere ir esta “libre”.



Tile estricto, movimiento por matriz

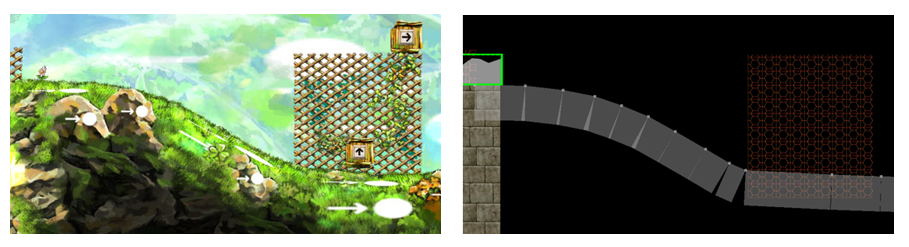
Una variación del sistema de tiles estricto inicial, es hacer que los elementos ocupen más de un tile, el sistema sigue siendo igual de sencillo de controlar, pero el movimiento parece más “libre”.



Mover los elementos por medio de las celdas de una matriz simplifica enormemente los cálculos (también facilita la lógica). Estos sistemas de movimiento fueron muy populares en juegos antiguos y aun hoy se siguen utilizando en muchos juegos móviles o “retro”.

**Mapas geométricos**

Otros juegos utilizan mapas donde no existe ninguna matriz, y los elementos pueden colocarse en cualquier parte, pudiendo admitir elementos con diferentes formas geométricas solo rectángulos.



En estos casos el cálculo de los movimientos y colisiones entre elementos se hace mucho más complejo (no es solo mirar si ciertas celdas están ocupadas). Suelen requerir muchos más cálculos y la mayor parte de los juegos implementa muchas más características físicas, como rebotes, diferentes materiales, fricción, etc. Resulta bastante complejo implementar estos algoritmos por ello gran parte de los juegos delegan estos cálculos en librerías externas como Box2D o Chimpunk (las usaremos en el Bloque)  
  
**Nuestra implementación**

Aunque la definición de nuestro nivel tiene pinta de ser un mapa de tiles lo hemos hecho así solo porque es mucho más fácil colocar los elementos que un mapa XML (sin usar un visualizador de mapas) nosotros vamos a implementar un **motor que maneje formas geométricas** (solo rectángulos).

|  |
| --- |
| **Nota:** tal y como definimos el mapa, “parece” que los elementos están colocados inicialmente en una matriz, pero:   * **NO vamos a usar un sistema de celdas para moverlos**, se pueden mover a cualquier X e Y disponible. * Los elementos del juego **NO tienen porque tener un tamaño proporcional a las celdas**, pueden tener cualquier dimensión (de hecho, el jugador, disparo y enemigo no son proporcionales a las celdas). |

Debemos implementar una clase que centralice el movimiento de todos los objetos de forma coordinada dependiendo de la velocidad que estos tengan y de las restricciones físicas (por ejemplo, otros elementos delante).

Comenzamos eliminando los movimientos de las **X** del **Jugador.js /** método **actualizar().**

|  |
| --- |
| actualizar(){  **this**.**animacion**.actualizar();   *// Establecer orientación* **if** ( **this**.**vx** > 0 ){  **this**.**orientacion** = ***orientaciones***.**derecha**;  }  **if** ( **this**.**vx** < 0 ){  **this**.**orientacion** = ***orientaciones***.**izquierda**;  }   *// Selección de animación* **switch** (**this**.**estado**){  **case *estados***.**disparando**:  **if** (**this**.**orientacion** == ***orientaciones***.**derecha**) {  **this**.**animacion** = **this**.**aDispararDerecha**;  }  **if** (**this**.**orientacion** == ***orientaciones***.**izquierda**) {  **this**.**animacion** = **this**.**aDispararIzquierda**;  }  **break**;  **case *estados***.**moviendo**:  **if** ( **this**.**vx** != 0 ) {  **if** (**this**.**orientacion** == ***orientaciones***.**derecha**) {  **this**.**animacion** = **this**.**aCorriendoDerecha**;  }  **if** (**this**.**orientacion** == ***orientaciones***.**izquierda**) {  **this**.**animacion** = **this**.**aCorriendoIzquierda**;  }  }  **if** ( **this**.**vx** == 0){  **if** (**this**.**orientacion** == ***orientaciones***.**derecha**) {  **this**.**animacion** = **this**.**aIdleDerecha**;  }  **if** (**this**.**orientacion** == ***orientaciones***.**izquierda**) {  **this**.**animacion** = **this**.**aIdleIzquierda**;  }  }  **break**;  }    *// Tiempo Disparo* **if** ( **this**.**tiempoDisparo** > 0 ) {  **this**.**tiempoDisparo**--;  }  **~~this~~**~~.~~**~~x~~** ~~=~~ **~~this~~**~~.~~**~~x~~** ~~+~~ **~~this~~**~~.~~**~~vx~~**~~;~~ **~~this~~**~~.~~**~~y~~** ~~=~~ **~~this~~**~~.~~**~~y~~** ~~+~~ **~~this~~**~~.~~**~~vy~~**~~;~~ } |

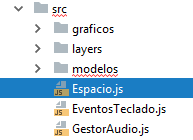
Eliminamos el movimiento de las **X** del **Enemigo.js** método **actualizar()**

|  |
| --- |
| actualizar (){  *// Actualizar animación* **this**.**animacion**.actualizar();   **switch** (**this**.**estado**){  **case *estados***.**moviendo**:  **this**.**animacion** = **this**.**aMover**;  **break**;  **case *estados***.**muriendo**:  **this**.**animacion** = **this**.**aMorir**;  **break**;  }   **if** ( **this**.**estado** != ***estados***.**muriendo**) {  **this**.**vx** = -1;  **~~this~~**~~.~~**~~x~~** ~~=~~ **~~this~~**~~.~~**~~x~~** ~~+~~ **~~this~~**~~.~~**~~vx~~**~~;~~  } } |

Eliminamos el movimiento de las **X** del **DisparoJugador.js** método **actualizar(),** no borrar el método actualizar() , dejarlo vacío para luego.

|  |
| --- |
| actualizar (){  **~~this~~**~~.~~**~~x~~** ~~=~~ **~~this~~**~~.~~**~~x~~** ~~+~~ **~~this~~**~~.~~**~~vx~~**~~;~~ } |

Creamos un nuevo fichero en **src/Espacio.js** , aquí incluiremos todo lo relativo al movimiento.



Creamos una clase **Espacio**, con un constructor que recibe el valor de la **gravedad**. (Los juegos con perspectiva frontal suelen tener una gravedad > 0 , los juegos con perspectiva aérea suelen tener gravedad = 0).

Creamos dos arrays donde vamos a incluir todos los elementos del espacio físico: **dinámicos** y **estáticos**.

* **Dinámicos** son los elementos que se pueden mover y que se van a ver afectados por la gravedad (jugador, enemigo…) .

En nuestro espacio dos elementos dinámicos si que pueden ocupar la misma superficie, esto se detectará con el metodillo **colisiona()** que ya tenemos implementado

* **Estáticos** van a ser los que ni se mueven ni se ven afectados por la gravedad (los bloques que forman el suelo y los muros).

Un elemento estático no va a poder ocupar la misma superficie que otro de cualquier tipo.

|  |  |
| --- | --- |
|  | **Rojo**. elementos dinámicos, Jugador y enemigo  **Gris**: Elementos estáticos, bloques. |

Creamos **src/Espacio.js .**

|  |
| --- |
| **class** Espacio {   constructor(gravedad) {  **this**.**gravedad** = gravedad;  **this**.**dinamicos** = [];  **this**.**estaticos** = [];  }   agregarCuerpoDinamico(modelo){  **this**.**dinamicos**.push(modelo);  }   agregarCuerpoEstatico(modelo){  **this**.**estaticos**.push(modelo);  }   eliminarCuerpoDinamico (modelo) {  **for** (**var** i = 0; i < **this**.**dinamicos**.length; i++) {  **if** (**this**.**dinamicos**[i] == modelo) {  **this**.**dinamicos**.splice(i, 1);  }  }  }    eliminarCuerpoEstatico(modelo){  **for** (**var** i = 0; i < **this**.**estaticos**.length; i++) {  **if** (**this**.**estaticos**[i] == modelo) {  **this**.**estaticos**.splice(i, 1);  }  }  }  } |

Agregamos el script **Espacio.js** al **Index.html**

|  |
| --- |
| <**canvas id="canvas" width="480" height="320"**></**canvas**>  <**script src="src/Espacio.js"**></**script**> <**script src="src/Globales.js"**></**script**> |

Ahora nos falta incluir una instancia de **Espacio** en el **GameLayer.js.** De momento sin gravedad para probar.

|  |
| --- |
| iniciar() {  *reproducirMusica*();  **this**.**espacio** = **new** Espacio(0);    **this**.**scrollX** = 0; |

Tenemos que incluir todos los elementos del Mapa en el **Espacio.** Desdeel método **cargarObjetoMapa()** introducimos el **jugador**, los **bloques** y los **enemigos**.

|  |
| --- |
| cargarObjetoMapa(simbolo, x, y){  **switch**(simbolo) {  **case "E"**:  **var** enemigo = **new** Enemigo(x,y);  enemigo.**y** = enemigo.**y** - enemigo.**alto**/2;  *// modificación para empezar a contar desde el suelo* **this**.**enemigos**.push(enemigo);  **this**.**espacio**.agregarCuerpoDinamico(enemigo);  **break**;  **case "1"**:  **this**.**jugador** = **new** Jugador(x, y);  *// modificación para empezar a contar desde el suelo* **this**.**jugador**.**y** = **this**.**jugador**.**y** - **this**.**jugador**.**alto**/2;  **this**.**espacio**.agregarCuerpoDinamico(**this**.**jugador**);  **break**;  **case "#"**:  **var** bloque = **new** Bloque(***imagenes***.**bloque\_tierra**, x,y);  bloque.**y** = bloque.**y** - bloque.**alto**/2;  *// modificación para empezar a contar desde el suelo* **this**.**bloques**.push(bloque);  **this**.**espacio**.agregarCuerpoEstatico(bloque);  **break**;  } } |

Los únicos elemento que no se crea en **cargarObjetoMapa**() son los **DisparoJugador**. Se crean en el método **procesarControles** del **GameLayer.**

|  |
| --- |
| procesarControles( ){  *// disparar* **if** ( ***controles***.**disparo** ){  **var** nuevoDisparo = **this**.**jugador**.disparar();  **if** ( nuevoDisparo != **null** ) {  **this**.**espacio**.agregarCuerpoDinamico(nuevoDisparo);  **this**.**disparosJugador**.push(nuevoDisparo);  }  } |

# Al igual que se añaden cuerpos al espacio, también se deben eliminar cuando ya no se van a usar (sino seguirán generando restricciones de movimiento).

El metodo **actualizar()** que comprueba las colisiones se encarga de eliminar los **Enemigos** y **DisparosJugador.**

El **Jugador** no llega a eliminarse nunca realmente, ya que al perder se “inicia” el nivel desde 0)

Hay que sacar los elementos del **Espacio** con **eliminarCuerpoDinamico** antes de borrarlos del array de lo contrarios e perderá la referencia a [i] / [j].

|  |
| --- |
| actualizar (){  **this**.**espacio**.actualizar();  **this**.**fondo**.**vx** = -1;  **this**.**fondo**.actualizar();  **this**.**jugador**.actualizar();   *// Eliminar disparos fuera de pantalla* **for** (**var** i=0; i < **this**.**disparosJugador**.length; i++){  **if** ( **this**.**disparosJugador**[i] != **null** &&  !**this**.**disparosJugador**[i].estaEnPantalla()){  **this**.**espacio** .eliminarCuerpoDinamico(**this**.**disparosJugador**[i]);  **this**.**disparosJugador**.splice(i, 1);  }  }    **for** (**var** i=0; i < **this**.**enemigos**.length; i++){  **this**.**enemigos**[i].actualizar();  }  **for** (**var** i=0; i < **this**.**disparosJugador**.length; i++) {  **this**.**disparosJugador**[i].actualizar();  }   *// colisiones* **for** (**var** i=0; i < **this**.**enemigos**.length; i++){  **if** ( **this**.**jugador**.colisiona(**this**.**enemigos**[i])){  **this**.iniciar();  }  }  *// colisiones , disparoJugador - Enemigo* **for** (**var** i=0; i < **this**.**disparosJugador**.length; i++){  **for** (**var** j=0; j < **this**.**enemigos**.length; j++){   **if** (**this**.**disparosJugador**[i] != **null** &&  **this**.**enemigos**[j] != **null** &&  **this**.**disparosJugador**[i].colisiona(**this**.**enemigos**[j])) {   **this**.**espacio** .eliminarCuerpoDinamico(**this**.**disparosJugador**[i]);  **this**.**disparosJugador**.splice(i, 1);  **this**.**enemigos**[j].**impactado**();  **this**.**puntos**.**valor**++;   }  }  }   *// Enemigos muertos fuera del juego* **for** (**var** j=0; j < **this**.**enemigos**.length; j++){  **if** ( **this**.**enemigos**[j] != **null** &&  **this**.**enemigos**[j].**estado** == ***estados***.**muerto** ) {   **this**.**espacio** .eliminarCuerpoDinamico(**this**.**enemigos**[j]);  **this**.**enemigos**.splice(j, 1);   }  }  } |

**Ir comprobando que no hay errores.** Aunque todavía no se va a mover nada, es recomendable comprobar que no hay errores en el código.

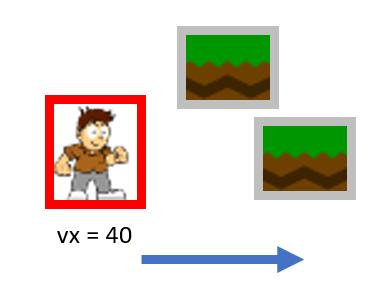


# Modificamos Espacio.js agregando un nuevo método actualizar(), el cual recorre todos los elementos dinámicos, a cada uno de esos elementos dinámicos:

1. Le aumentamos la velocidad **vy** en función de la **gravedad.** Impidiendo que pase de 20 (Siempre hay que poner límites a las velocidades que aumentan o disminuyen de forma constante, sino pueden llegar a valores demasiado extremos)
2. Evaluamos si debe moverse hacia la derecha **moverDerecha** o la izquierda **moverIzquierda**.

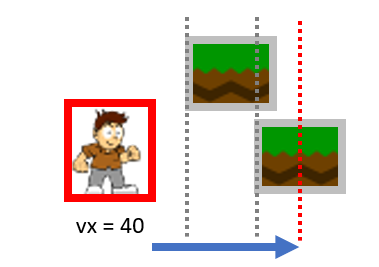
**moverDerecha**

* Hay que calcular el **movimientoPosible**, que el elemento dinámico tenga velocidad 40 no significa que se vaya a mover 40, puede tener un objeto **estático** delante.



* Partimos de que el **movimientoPosible** “ideal” es la velocidad, pero debemos comprobar si hay algo delante.   
  Para ello recorremos TODOS los cuerpos estáticos que
  + Están a nuestra derecha. derechaDinamico + vx > IzquierdaEstatico

derechaDinamico + vx es la línea roja.



* + Están a nuestra altura (para ese bloque nos afecte y haya choque – **2 condiciones**):   
     1) abajoDinamico > arribaEstatico

|  |  |
| --- | --- |
|  |  |

2) arribaDinamico < abajoEstatico

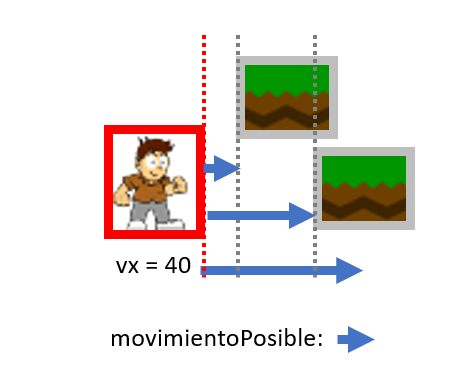
|  |  |
| --- | --- |
|  |  |

* Calculamos la distancia respecto a **TODOS** esos elementos **estáticos** que cumplen las condiciones.

La distancia se calcula: izquierdaEstatico - derechaDinamico

Sí el **movimientoPosible** que tenemos guardado **es más grande** que la distancia lo desechamos! y actualizamos el nuevo **movimientoPosible**

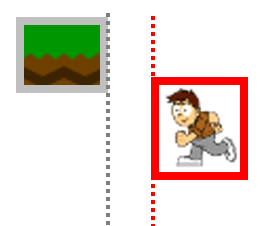
Vamos DESCARTANDO siempre **movimientoPosible** más grande para finalmente quedarnos con el más pequeño.



**moverIzquierda**

Lo mismo, pero hacia el otro lado, tenemos en cuenta

* Va hacia la izquierda, consideramos el **ladoIzquierdo** del elemento **dinamico** y el **ladoDerecho** del **estatico**.



* Hacia la izquierda los **movimientoPosibles** son **negativos**, por lo tanto, desechamos el **movimientoPosible** actual **si** **es más pequeño** que la distancia al obstáculo ( -10 es más pequeño que -3, pero realmente es moverse más).

|  |
| --- |
| actualizar(){  **for**( **var** i=0; i < **this**.**dinamicos**.length; i++){   *// aplicar gravedad ( dinamicos)* **this**.**dinamicos**[i].**vy** = **this**.**dinamicos**[i].**vy** + **this**.**gravedad**;  *// maxima velocidad de caida por gravedad* **if** (**this**.**dinamicos**[i].**vy** > 20) {  **this**.**dinamicos**[i].**vy** = 20;  }   *//derecha* **this**.moverDerecha(i);  **this**.moverIzquierda(i);  }  }  moverDerecha(i){  **if** ( **this**.**dinamicos**[i].**vx** > 0){  **var** movimientoPosible = **this**.**dinamicos**[i].**vx**;  *// El mejor "idealmente" vx partimos de ese* **for**(**var** j=0; j < **this**.**estaticos**.length; j++){  **var** derechaDinamico  = **this**.**dinamicos**[i].**x** + **this**.**dinamicos**[i].**ancho**/2;  **var** arribaDinamico  = **this**.**dinamicos**[i].**y** - **this**.**dinamicos**[i].**alto**/2;  **var** abajoDinamico  = **this**.**dinamicos**[i].**y** + **this**.**dinamicos**[i].**alto**/2;  **var** izquierdaEstatico  = **this**.**estaticos**[j].**x** - **this**.**estaticos**[j].**ancho**/2;  **var** arribaEstatico  = **this**.**estaticos**[j].**y** - **this**.**estaticos**[j].**alto**/2;  **var** abajoEstatico  = **this**.**estaticos**[j].**y** + **this**.**estaticos**[j].**alto**/2;   *// Alerta!, Elemento estático en la trayectoria.* **if** ( (derechaDinamico + **this**.**dinamicos**[i].**vx**) >= izquierdaEstatico  && derechaDinamico <= izquierdaEstatico  && arribaEstatico < abajoDinamico  && abajoEstatico > arribaDinamico){   *// Comprobamos si la distancia al estático es menor  // que nuestro movimientoPosible actual* **if** (movimientoPosible >= izquierdaEstatico - derechaDinamico){  *// La distancia es MENOR que nuestro movimiento posible  // Tenemos que actualizar el movimiento posible a uno menor* movimientoPosible = izquierdaEstatico - derechaDinamico ;  }   }   }  *// Ya se han comprobado todos los estáticos* **this**.**dinamicos**[i].**x** = **this**.**dinamicos**[i].**x** + movimientoPosible;  **this**.**dinamicos**[i].**vx** = movimientoPosible;  } } |

Método **moverIzquierda().**

|  |
| --- |
| moverIzquierda(i){  *// Izquierda* **if** ( **this**.**dinamicos**[i].**vx** < 0){  **var** movimientoPosible = **this**.**dinamicos**[i].**vx**;  *// El mejor "idealmente" vx partimos de ese* **for**(**var** j=0; j < **this**.**estaticos**.length; j++){  **var** izquierdaDinamico  = **this**.**dinamicos**[i].**x** - **this**.**dinamicos**[i].**ancho**/2;  **var** arribaDinamico  = **this**.**dinamicos**[i].**y** - **this**.**dinamicos**[i].**alto**/2;  **var** abajoDinamico  = **this**.**dinamicos**[i].**y** + **this**.**dinamicos**[i].**alto**/2;  **var** derechaEstatico  = **this**.**estaticos**[j].**x** + **this**.**estaticos**[j].**ancho**/2;  **var** arribaEstatico  = **this**.**estaticos**[j].**y** - **this**.**estaticos**[j].**alto**/2;  **var** abajoEstatico  = **this**.**estaticos**[j].**y** + **this**.**estaticos**[j].**alto**/2;    *// Alerta!, Elemento estático en la trayectoria.* **if** ( (izquierdaDinamico + **this**.**dinamicos**[i].**vx**) <= derechaEstatico  && izquierdaDinamico >= derechaEstatico  && arribaEstatico < abajoDinamico  && abajoEstatico > arribaDinamico ){   *// Comprobamos si la distancia al estático es mayor  // que nuestro movimientoPosible actual* **if** (movimientoPosible <= derechaEstatico - izquierdaDinamico ){  *// La distancia es MAYOR que nuestro movimiento posible  // Tenemos que actualizar el movimiento posible a uno mayor* movimientoPosible = derechaEstatico - izquierdaDinamico ;  }   }  }   *// Ya se han comprobado todos los estaticos* **this**.**dinamicos**[i].**x** = **this**.**dinamicos**[i].**x** + movimientoPosible;  **this**.**dinamicos**[i].**vx** = movimientoPosible; }  } |

Volvemos al **Gamelayer.js** y nos colocamos en el método **actualizar()** incluimos una llamada a **this.espacio.actualizar()**.

|  |
| --- |
| actualizar (){  **this**.**espacio**.actualizar();  **this**.**fondo**.**vx** = -1;  **this**.**fondo**.actualizar(); |

Probamos el juego, debería dejarnos mover el **Jugador** hacia la derecha y la izquierda.

Probamos a modificar el mapa **0.txt**, podemos editarlo para colocar bloques estáticos delante.

|  |  |
| --- | --- |
|  |  |

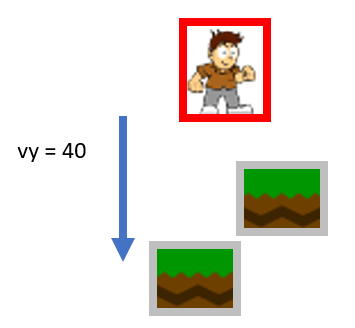
# Espacio físico - movimiento Y

Continuamos ampliando la clase **Espacio,** en su método **actualizar()** añadimos llamadas a **moverArriba(i)** y **moverAbajo(i)**

|  |
| --- |
| actualizar(){  **for**( **var** i=0; i < **this**.**dinamicos**.length; i++){   *// aplicar gravedad ( dinamicos)* **this**.**dinamicos**[i].**vy** = **this**.**dinamicos**[i].**vy** + **this**.**gravedad**;  *// maxima velocidad de caida por gravedad* **if** (**this**.**dinamicos**[i].**vy** > 20) {  **this**.**dinamicos**[i].**vy** = 20;  }   *//derecha* **this**.moverDerecha(i);  **this**.moverIzquierda(i);  **this**.moverArriba(i);  **this**.moverAbajo(i);  } } |

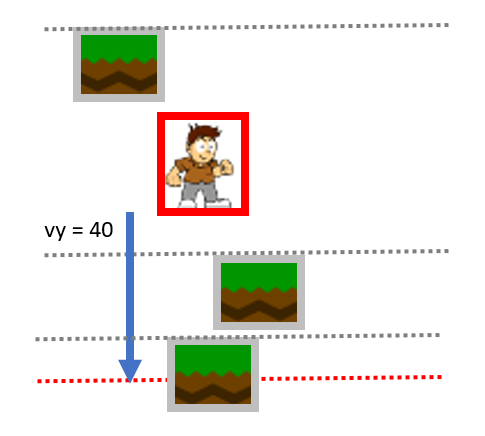
Implementamos el método **moverAbajo(i)**

* Hay que calcular el **movimientoPosible**, que el elemento dinámico tenga velocidad 40 no significa que se vaya a mover 40, puede tener un objeto **estático** delante.

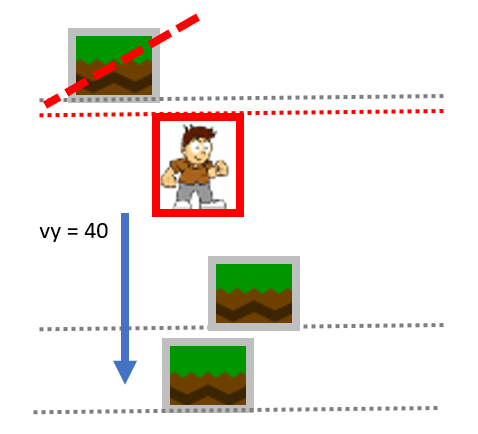


* Partimos de que el **movimientoPosible** “ideal” es la velocidad, pero debemos comprobar si hay algo debajo.   
  Para ello recorremos TODOS los cuerpos estáticos que
  + Están debajo

abajoDinamico + vel > arribaEstatico



También arribaDinamico < abajoEstatico (no nos sobrepasan)



* + Están a nuestra anchura (sino no hay choque) :   
     1) izquierdaDinamico < derechaEstatico

|  |  |
| --- | --- |
|  |  |

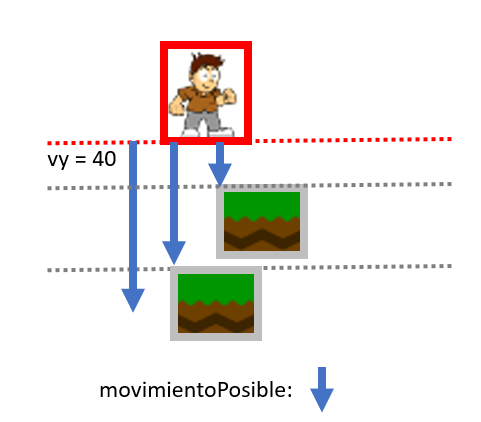
2) derechaDinamico > izquierdaEstatico

|  |  |
| --- | --- |
|  |  |

* Calculamos la distancia respecto a **TODOS** esos elementos **estáticos** que cumplen las condiciones. arribaEstatico - abajoDinamico

Sí el **movimientoPosible** que tenemos guardado **es más grande** que la distancia lo desechamos! y actualizamos el nuevo **movimientoPosible**

Vamos desechando siempre **movimientoPosible** más grande lo que hace que al final nos quedemos con el más pequeño.

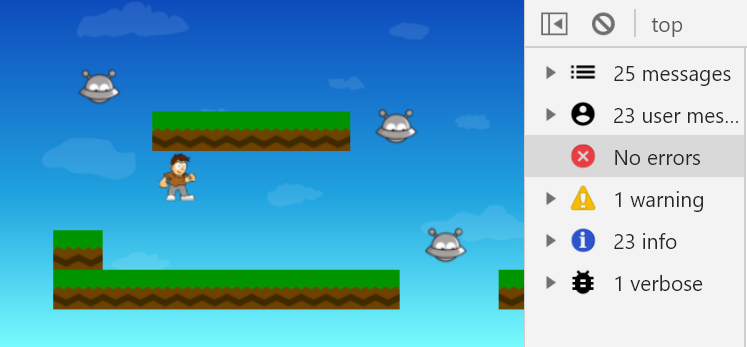


|  |
| --- |
| moverAbajo(i){  **if** ( **this**.**dinamicos**[i].**vy** > 0){  **var** movimientoPosible = **this**.**dinamicos**[i].**vy**;  *// El mejor "idealmente" es la velocidad vy.* **for**(**var** j=0; j < **this**.**estaticos**.length; j++){  **var** arribaDinamico  = **this**.**dinamicos**[i].**y** - **this**.**dinamicos**[i].**alto**/2;  **var** abajoDinamico  = **this**.**dinamicos**[i].**y** + **this**.**dinamicos**[i].**alto**/2;  **var** derechaDinamico  = **this**.**dinamicos**[i].**x** + **this**.**dinamicos**[i].**ancho**/2;  **var** izquierdaDinamico  = **this**.**dinamicos**[i].**x** - **this**.**dinamicos**[i].**ancho**/2;  **var** arribaEstatico  = **this**.**estaticos**[j].**y** - **this**.**estaticos**[j].**alto**/2;  **var** abajoEstatico  = **this**.**estaticos**[j].**y** + **this**.**estaticos**[j].**alto**/2;  **var** derechaEstatico  = **this**.**estaticos**[j].**x** + **this**.**estaticos**[j].**ancho**/2;  **var** izquierdaEstatico  = **this**.**estaticos**[j].**x** - **this**.**estaticos**[j].**ancho**/2;   *// Alerta!, Elemento estático en la trayectoria.* **if** ( (abajoDinamico + **this**.**dinamicos**[i].**vy**) >= arribaEstatico &&  arribaDinamico < abajoEstatico  && izquierdaDinamico < derechaEstatico  && derechaDinamico > izquierdaEstatico ){   *// Comprobamos si la distancia al estático es menor  // que nuestro movimientoPosible actual* **if** (movimientoPosible >= arribaEstatico - abajoDinamico ){  *// La distancia es MENOR que nuestro movimiento posible  // Tenemos que actualizar el movimiento posible a uno menor* movimientoPosible = arribaEstatico - abajoDinamico ;  }  }  }   *// Ya se han comprobado todos los estáticos* **this**.**dinamicos**[i].**y** = **this**.**dinamicos**[i].**y** + movimientoPosible;  **this**.**dinamicos**[i].**vy** = movimientoPosible;  } } |

Finalmente implementamos **moverArriba(),** con un razonamiento casi idéntico al anterior, pero teniendo en cuenta el lado de debajo del cuerpo dinámico y que ahora el movimientoPosible es negativo.

|  |
| --- |
| moverArriba(i){  **if** ( **this**.**dinamicos**[i].**vy** < 0){  **var** movimientoPosible = **this**.**dinamicos**[i].**vy**;  *// El mejor "idealmente" es la velocidad vy.* **for**(**var** j=0; j < **this**.**estaticos**.length; j++){  **var** arribaDinamico  = **this**.**dinamicos**[i].**y** - **this**.**dinamicos**[i].**alto**/2;  **var** abajoDinamico  = **this**.**dinamicos**[i].**y** + **this**.**dinamicos**[i].**alto**/2;  **var** derechaDinamico  = **this**.**dinamicos**[i].**x** + **this**.**dinamicos**[i].**ancho**/2;  **var** izquierdaDinamico  = **this**.**dinamicos**[i].**x** - **this**.**dinamicos**[i].**ancho**/2;  **var** arribaEstatico  = **this**.**estaticos**[j].**y** - **this**.**estaticos**[j].**alto**/2;  **var** abajoEstatico  = **this**.**estaticos**[j].**y** + **this**.**estaticos**[j].**alto**/2;  **var** derechaEstatico  = **this**.**estaticos**[j].**x** + **this**.**estaticos**[j].**ancho**/2;  **var** izquierdaEstatico  = **this**.**estaticos**[j].**x** - **this**.**estaticos**[j].**ancho**/2;   *// Alerta!, Elemento estático en la trayectoria* **if** ( (arribaDinamico + **this**.**dinamicos**[i].**vy**) <= abajoEstatico &&  abajoDinamico > arribaEstatico  && izquierdaDinamico < derechaEstatico  && derechaDinamico > izquierdaEstatico ){   *// Comprobamos si la distancia al estático es MAYOR  // que nuestro movimientoPosible actual* **if** (movimientoPosible <= abajoEstatico - arribaDinamico ){  *// La distancia es MAYOR que nuestro movimiento posible  // Tenemos que actualizar el movimiento posible a uno mayor* movimientoPosible = abajoEstatico - arribaDinamico ;  }   }  }   **this**.**dinamicos**[i].**y** = **this**.**dinamicos**[i].**y** + movimientoPosible;  **this**.**dinamicos**[i].**vy** = movimientoPosible;  } } |

Abrimos la aplicación y comprobamos que el funcionamiento es correcto y que no tiene errores.



Deberíamos comprobar que el movimiento en el eje **Y** y el choque con los bloques es correcto en todos los casos (para ello podemos modificar el mapa).

|  |  |
| --- | --- |
|  |  |

# Jugador – Saltar

La configuración actual del **Espacio** (sin gravedad) y **el sistema de control en el eje Y** del **Jugador** podría servir para un juego con vista aérea, pero **NO** para uno con vista lateral.

Comenzamos accediendo al método **iniciar()** del **GameLayer.js** y modificando la creación del **Espacio** para que tenga 1 de gravedad.

|  |
| --- |
| iniciar() {  *reproducirMusica*();  **this**.**espacio** = **new** Espacio(1); |

Ahora ya hay gravedad, pero el **usuario** sigue pudiendo mover al jugador hacia arriba y abajo cuando quiere.

* **Haría arriba:** Al saltar el jugador recibe una **vy < 0** (recordamos que el eje negativo está arriba)
* **Hacia abajo:** El jugador NO debe poder moverse hacia abajo deliberadamente, la **gravedad** se encarga de empujarle

Modificamos el movimiento Y en el método **procesarControles().** Dejamos de utilizar la función **moverY,** únicamente cuandonos llega la señal de mover hacia arriba **controles.moverY > 0** ejecutamos el **this.jugador.saltar()**

|  |
| --- |
| *// Eje Y* **if** ( ***controles***.**moverY** > 0 ){  **this**.**jugador**.saltar();  **~~this~~**~~.~~**~~jugador~~**~~.moverY(-1);~~   } **else if** ( ***controles***.**moverY** < 0 ){  **~~this~~**~~.~~**~~jugador~~**~~.moverY(1);~~   } **else** {  **~~this~~**~~.~~**~~jugador~~**~~.moverY(0);~~  } } |

Abrimos la clase **Jugador** e implementamos **saltar().** Este método va a tener inicialmente una lógica simple.

|  |
| --- |
| saltar(){  **this**.**vy** = -16; } |

Probamos el juego. Si siempre que se pulsa el control el **jugador** salta el movimiento no va a quedar bien.



Los juegos que tratan de imitar una “física realista” solo dejan saltar cuando NO estamos en el aire.

|  |
| --- |
| saltar(){  **if** ( !**this**.**enElAire** ) {  **this**.**vy** = -16;  **this**.**enElAire** = **true**;  } } |

Pero realmente el jugador puede estar **enElAire = true** por dos motivos:

1. Ya contemplado: ha saltado, método **saltar().**
2. Sin contemplar aun: Se está cayendo de una plataforma.

Vamos a utilizar una variable **choqueAbajo**, que nos dirá si el jugador tiene un bloque estático debajo, “lo está pisando”.

De momento utilizamos la variable **choqueAbajo** como si funcionase correctamente, pero debe ser el espacio de Físicas quien se encargue de darle valor en cada iteración.

|  |
| --- |
| actualizar(){  **this**.**animacion**.actualizar();    *// ¿Esta en el aire?* **if** (**this**.**choqueAbajo** == **true**){  **this**.**enElAire** = **false**;  } **else** {  **this**.**enElAire** = **true**;  } |

Accedemos al **Espacio.js** para que nos avise cuando cualquier elemento dinámico choca con uno estático por abajo.

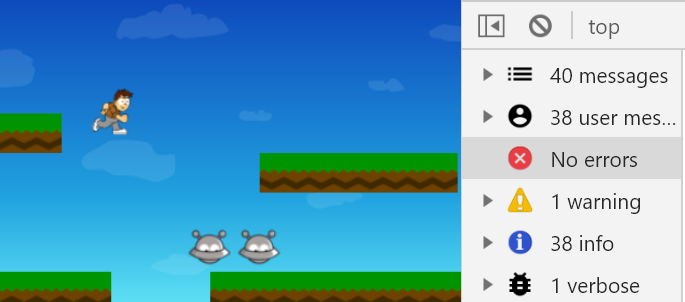
Primero editamos el método **actualizar(),** suponemos inicialmente que **NO** hay ningún **choqueAbajo**, igualando las variables a false

|  |
| --- |
| actualizar(){  **for**( **var** i=0; i < **this**.**dinamicos**.length; i++){   *// aplicar gravedad ( dinámicos)* **this**.**dinamicos**[i].**vy** = **this**.**dinamicos**[i].**vy** + **this**.**gravedad**;  *// maxima velocidad de caída por gravedad* **if** (**this**.**dinamicos**[i].**vy** > 20) {  **this**.**dinamicos**[i].**vy** = 20;  }   *// reiniciar choques* **this**.**dinamicos**[i].**choqueAbajo** = **false**;    *//derecha* **this**.moverDerecha(i);  . . . |

El método **moverAbajo(i),** es el que realmente sabe si ha habido un **choqueAbajo**.

|  |
| --- |
| *// Alerta!, Elemento estático en la trayectoria.* **if** ( (abajoDinamico + **this**.**dinamicos**[i].**vy**) >= arribaEstatico &&  arribaDinamico < abajoEstatico  && izquierdaDinamico < derechaEstatico  && derechaDinamico > izquierdaEstatico ){   *// Comprobamos si la distancia al estático es menor  // que nuestro movimientoPosible actual* **if** (movimientoPosible >= arribaEstatico - abajoDinamico ){  *// La distancia es MAYOR que nuestro movimiento posible  // Tenemos que actualizar el movimiento posible a uno mayor* movimientoPosible = arribaEstatico - abajoDinamico ;  **this**.**dinamicos**[i].**choqueAbajo** = **true**; |

Probamos el juego y comprobamos que solo nos permite saltar cuando tenemos un bloque debajo.



# Hacer desaparecer los disparos (Detección de “choque” simple)

Si hemos probado a disparar podemos ver cuando el **disparoJugador** “choca” contra un **bloque estático** se queda sin velocidad, pasa a tener **vx=0** y permanece suspendido en el aire.



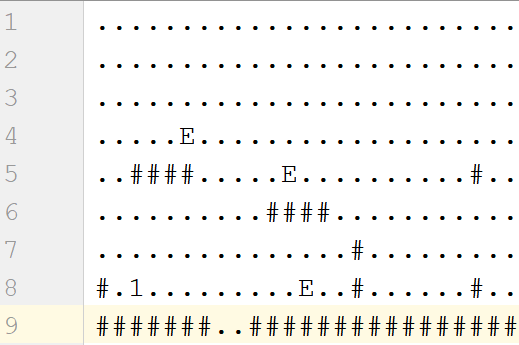
En el **actualizar()** del **GameLayer** podemos directamente eliminar del juego **disparoJugador** que tienen **vx == 0**. Recordamos que también deben ser eliminados del espacio físico.

|  |
| --- |
| actualizar (){  **this**.**espacio**.actualizar();  **this**.**fondo**.**vx** = -1;  **this**.**fondo**.actualizar();  **this**.**jugador**.actualizar();   *// Eliminar disparos sin velocidad* **for** (**var** i=0; i < **this**.**disparosJugador**.length; i++){  **if** ( **this**.**disparosJugador**[i] != **null** &&  **this**.**disparosJugador**[i].**vx** == 0){    **this**.**espacio** .eliminarCuerpoDinamico(**this**.**disparosJugador**[i]);  **this**.**disparosJugador**.splice(i, 1);  }  } |

|  |
| --- |
| **Nota:** Este mecanismo es un **muy básico** y probablemente NO nos sirva para otros elementos diferentes al disparo, más adelante implementaremos otro mecanismo para detectar “choques” con elementos estáticos. |

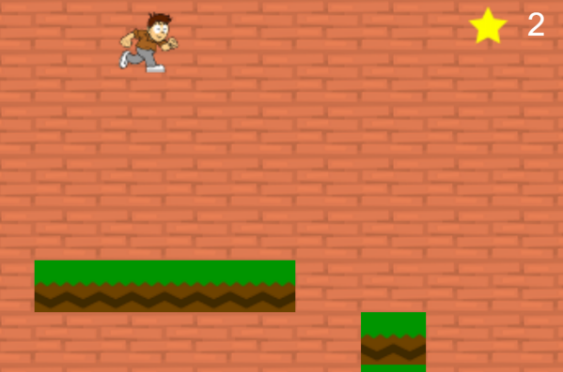
# \*Tiles como fondo (Para casa, hacer y luego borrar) Muchos juegos con vista área utilizan Bloques/Tiles para los fondos o terrenos del mapa (estos juegos no suelen usar un fondo externo al mapa). Aunque este tipo de juegos suelen construirse con mapas multicapa (permiten colocar más de un elemento en la misma posición, por ejemplo “hierba” una “moneda” y un “enemigo”) también pueden construirse con mapas que poseen una única capa (como el nuestro).

  
Vamos a suponer que los símbolos **‘ . ’** del mapa representan un **bloque de fondo de ladrillos** (para que un bloque sea “simplemente un fondo” basta con NO añadirlo al espacio físico.



**Adaptación:** Como se trata de un mapa de una sola capa **debemos suponer que detrás de cualquier símbolo 1 (jugador) y E (Enemigo) también hay un ‘ . ’ ,** . Si no colocamos un fondo detrás de ellos cuando se muevan quedará un “espacio vacío”.

|  |
| --- |
| cargarObjetoMapa(simbolo, x, y){  **switch**(simbolo) {  **case "."**:  **var** bloque = **new** Bloque(***imagenes***.**bloque\_fondo\_muro**, x,y);  bloque.**y** = bloque.**y** - bloque.**alto**/2;  *// modificación para empezar a contar desde el suelo* **this**.**bloques**.push(bloque);  **break**;  **case "E"**:  **var** enemigo = **new** Enemigo(x,y);  enemigo.**y** = enemigo.**y** - enemigo.**alto**/2;  *// modificación para empezar a contar desde el suelo* **this**.**enemigos**.push(enemigo);  **this**.**espacio**.agregarCuerpoDinamico(enemigo);  **var** bloque = **new** Bloque(***imagenes***.**bloque\_fondo\_muro**, x,y);  bloque.**y** = bloque.**y** - bloque.**alto**/2;  *// modificación para empezar a contar desde el suelo* **this**.**bloques**.push(bloque);  **break**;  **case "1"**:  **this**.**jugador** = **new** Jugador(x, y);  *// modificación para empezar a contar desde el suelo* **this**.**jugador**.**y** = **this**.**jugador**.**y** - **this**.**jugador**.**alto**/2;  **this**.**espacio**.agregarCuerpoDinamico(**this**.**jugador**);  **var** bloque = **new** Bloque(***imagenes***.**bloque\_fondo\_muro**, x,y);  bloque.**y** = bloque.**y** - bloque.**alto**/2;  *// modificación para empezar a contar desde el suelo* **this**.**bloques**.push(bloque);  **break**;  **case "#"**:  **var** bloque = **new** Bloque(***imagenes***.**bloque\_tierra**, x,y);  bloque.**y** = bloque.**y** - bloque.**alto**/2;  *// modificación para empezar a contar desde el suelo* **this**.**bloques**.push(bloque);  **this**.**espacio**.agregarCuerpoEstatico(bloque);  **break**;  } } |



# Enemigo – movimiento con rebotes (sin predicción de caída) Para lograr que el Enemigo “rebote” cuando se encuentre con un bloque crearemos una variable vxInteligencia , esa será la velocidad “ideal” que su inteligencia le manda tener (aunque los bloques pueden frenarle). El motor de físicas no usa vxInteligencia para mover al enemigo simplemente la usamos en su lógica de negocio para “compara” la velocidad que le gustaría tener con la que realmente tiene vx.

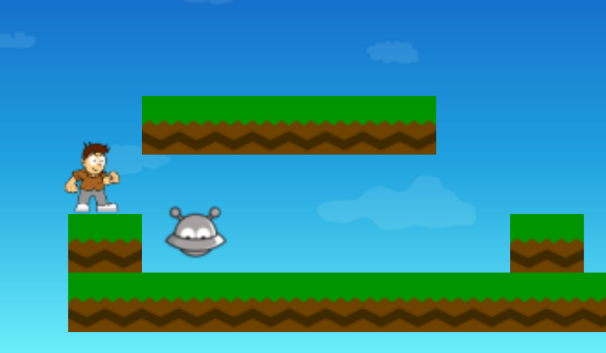
|  |
| --- |
| constructor(x, y) {  **super**(***imagenes***.**enemigo**, x, y)  **this**.**estado** = ***estados***.**moviendo**;  **this**.**vxInteligencia** = -1;  **this**.**vx** = **this**.**vxInteligencia**; |

Modificamos el método **actualizar().**

* Sí está en **estados.muriendo** su **vx** tiene que ser **0.**
* En caso de que no esté muriendo y su **vx== 0** significa que ha debido chocar, invertimos la **vxInteligencia** (que vaya hacia la otra dirección) y se la asignamos a su **vx**.

|  |
| --- |
| actualizar (){  *// Actualizar animación* **this**.**animacion**.actualizar();   **switch** (**this**.**estado**){  **case *estados***.**moviendo**:  **this**.**animacion** = **this**.**aMover**;  **break**;  **case *estados***.**muriendo**:  **this**.**animacion** = **this**.**aMorir**;  **break**;  }   **if** ( **this**.**estado** == ***estados***.**muriendo**) {  **this**.**vx** = 0;  } **else** {  **if** ( **this**.**vx** == 0){  **this**.**vxInteligencia** = **this**.**vxInteligencia** \* -1;  **this**.**vx** = **this**.**vxInteligencia**;  }  } } |

Editamos el mapa para colocar un **enemigo** entre dos **bloques**, de esta forma podemos probar el rebote.



|  |
| --- |
| **Alternativa:** De una forma muy similar a la que implementamos **choqueAbajo** podemos implementar **choqueDerecha** y **choqueIzquierda** dentro del motor de movimiento.  De esta forma todos los elementos dentro del espacio de físicas podrían saber rápidamente si han chocado por algún lado**.** |

# Jugador – Caer fuera del mapa

Cuando el **jugador** abandona la pantalla por la parte inferior el juego no se acaba, vamos a hacer que en cada iteración del juego se compruebe la posición y del **jugador** y que si esta supera un límite, en este caso se inicia de nuevo el nivel.

Abrimos **GameLayer.js** y añadimos la comprobación en el método **actualizar()**.

|  |
| --- |
| actualizar (){  *// Jugador se cae* **if** ( **this**.**jugador**.**y** > 480 ){  **this**.iniciar();  }   **this**.**espacio**.actualizar(); |

# \*Vidas y jugador invulnerable (Para hacer en casa)

Vamos a implementar que el **jugador** disponga de 3 vidas para superar un nivel. Por lo tanto, cada vez que se instancia el **jugador** sus **vidas** serán 3.

Accedemos a **Jugador** y modificamos su **constructor**. Aprovechamos también para incluir una variable global **tiempoInvulnerable** (cuando está variable valga más de 0 el jugador no podrá perder vidas).

|  |
| --- |
| **class** Jugador **extends** Modelo {   constructor(x, y) {  **super**(***imagenes***.**jugador** , x, y)  **this**.**vidas** = 3;  **this**.**tiempoInvulnerable** = 0; |

Implementamos un nuevo método **glopeado(),** reducirá una **vida** (siempre que le queden) e colocará el **this.tiempoInvulnerable** a 100.

|  |
| --- |
| golpeado (){  **if** (**this**.**tiempoInvulnerable** <= 0) {  **if** (**this**.**vidas** > 0) {  **this**.**vidas**--;  **this**.**tiempoInvulnerable** = 100;  *// 100 actualizaciones de loop* }  } } |

En el **actualizar()** si el **this.tiempoInvulnerable** tiene un valor mayor a 0 debemos ir reduciéndolo en cada iteración.

|  |
| --- |
| actualizar(){  **if** (**this**.**tiempoInvulnerable** > 0 ){  **this**.**tiempoInvulnerable** --;  }    **this**.**animacion**.actualizar(); |

Aunque no es necesario para el funcionamiento podemos modificar el **dibujar()**para que se pinte de forma semitransparente cuando esta invulnerable (estableciendo el alpha del canvas al 0.5 ).

|  |
| --- |
| dibujar (scrollX){  scrollX = scrollX || 0;  **if** ( **this**.**tiempoInvulnerable** > 0) {  ***contexto***.**globalAlpha** = 0.5;  **this**.**animacion**.dibujar(**this**.**x** - scrollX, **this**.**y**);  ***contexto***.**globalAlpha** = 1;  } **else** {  **this**.**animacion**.dibujar(**this**.**x** - scrollX, **this**.**y**);  } } |

Ahora modificamos el **GameLayer.js** , el nivel NO debe restablecerse siempre que colisionan **Jugador** y el **Enemigo**, se debe llamar a **jugador.golpeado()** y reiniciar solo si el jugador se queda sin vidas.

El método **actualizar()** del **GameLayer** era el que calculaba todas las colisiones, incluimos la nueva lógica.

|  |
| --- |
| *// colisiones* **for** (**var** i=0; i < **this**.**enemigos**.length; i++){  **if** ( **this**.**jugador**.colisiona(**this**.**enemigos**[i])){  **~~this~~**~~.iniciar();~~  **this**.**jugador**.golpeado();  **if** (**this**.**jugador**.**vidas** <= 0){  **this**.iniciar();  }  } } |

Comprobamos que cuando los enemigos nos golpean el jugador se vuelve invulnerable y que al recibir 3 impactos se reinicia el nivel.



|  |
| --- |
| **Idea**. Podríamos incluir un interfaz para mostrar las vidas en pantalla:   * **Texto**, de forma casi idéntica a como incluimos el contador de puntos. * **Imágenes**, con iconos de vidas y pintarlos solo cuando el jugador tiene X vidas, Ejemplo: el icono vida1 solo se pinta si tiene 1 vida o más, el icono vida2 solo se pinta si tiene 2 vidas o más, etc. |

# \*Jugador - Animación de saltar (realmente “saltar” y “caer”) Para hacer en casa

Abrimos el fichero **jugador.js** y agregamos dos nuevas animaciones en el constructor, la animación de salto se reproducirá de forma infinita mientras este en el aire (infinita: no hace falta enviar un callback como parámetro para indicar cuando acaba la animación)



|  |
| --- |
| **this**.**aCorriendoDerecha** =  **new** Animacion(***imagenes***.**jugador\_corriendo\_derecha**,  **this**.**ancho**, **this**.**alto**, 8, 8); **this**.**aCorriendoIzquierda** = **new** Animacion(***imagenes***.**jugador\_corriendo\_izquierda**,  **this**.**ancho**, **this**.**alto**, 8, 8); **this**.**aSaltandoDerecha** = **new** Animacion(***imagenes***.**jugador\_saltando\_derecha**,  **this**.**ancho**, **this**.**alto**, 6, 4); **this**.**aSaltandoIzquierda** = **new** Animacion( ***imagenes***.**jugador\_saltando\_izquierda**,  **this**.**ancho**, **this**.**alto**, 6, 4);  **this**.**animacion** = **this**.**aIdleDerecha**; |

Al estado de **saltando** se accederá desde el estado de **movimiento** (no vamos tenemos en cuenta saltando para otros estamos como el de **disparando**, este tiene más prioridad y se pone en estado disparando independientemente de en el estado en el que estuviésemos antes).  
  
Para determinar el **estado = estado.saltando,** solo hay que fijarse en que el estado actual sea: **estado.movimiento** y en el valor de **enElAire**.

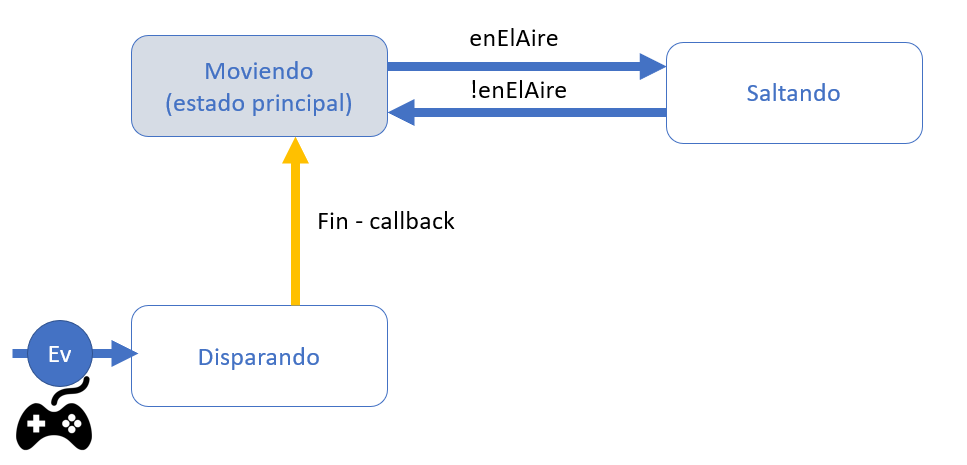
|  |
| --- |
| actualizar(){  **if** (**this**.**enElAire** && **this**.**estado** == ***estados***.**moviendo** ){  **this**.**estado** = ***estados***.**saltando**;  }  **if** (!**this**.**enElAire** && **this**.**estado** == ***estados***.**saltando** ){  **this**.**estado** = ***estados***.**moviendo**;  }   **if** (**this**.**tiempoInvulnerable** > 0 ){  **this**.**tiempoInvulnerable**--;  } |

Dentro del switch de **actualizar()** debemos agregar también un case nuevo para **estados.saltando**

|  |
| --- |
| *// Selección de animación* **switch** (**this**.**estado**){  **case *estados***.**saltando**:  **if** (**this**.**orientacion** == ***orientaciones***.**derecha**){  **this**.**animacion** = **this**.**aSaltandoDerecha**;  }  **if** (**this**.**orientacion** == ***orientaciones***.**izquierda**){  **this**.**animacion** = **this**.**aSaltandoIzquierda**;  }  **break**;  **case *estados***.**disparando**:  **if** (**this**.**orientacion** == ***orientaciones***.**derecha**) {  **this**.**animacion** = **this**.**aDispararDerecha**;  }  **if** (**this**.**orientacion** == ***orientaciones***.**izquierda**) {  **this**.**animacion** = **this**.**aDispararIzquierda**;  }  **break**; |

El diagrama de estados de la animación del jugador queda de la siguiente manera.

* Podemos pasar de **Moviendo** a **Saltando** si estamos en el aire
* Al acabar **Saltando** se vuelve a **Moviendo**.
* En cualquier el jugador puede cambiar al estado **Disparando**



|  |
| --- |
| **Alternativa:** El estado **Saltando** podría haberse suprimido y ser solo un caso especial de **Moviendo**, además de Idle y camina el estado Moviendo podría controlar el salto mirando el primer lugar el valor de la variable “enElAire”. |

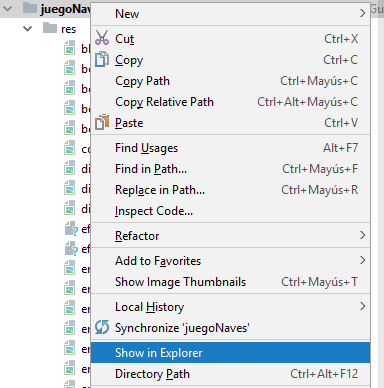
# Entrenamiento

Antes de continuar con el guion vamos a proponer algunas modificaciones para entrenarnos

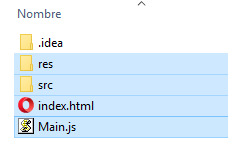
Debemos crear un **proyecto nuevo** y **copiar el contenido a nivel de fichero** de este en el otro para no estropearlo. **En las siguientes sesiones necesitamos el proyecto sin ampliaciones**

|  |
| --- |
| **Evitar tener proyectos con el mismo nombre.** Evitamos duplicar la carpeta entera de un proyecto, porque tener 2 proyectos con el mismo nombre suele dar problemas de CACHE en el navegador (carga uno cuando debería cargar otro). |

Para ver el proyecto en el explorador de Windows pulsamos botón derecho y **Show in Explorer**



Copiamos los **ficheros de implementación** en el nuevo proyecto.



**Bloques destruibles con disparos**

Incluye unos nuevos tipos de bloques que desaparezcan cuando el jugador les dispara.