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

|  |  |
| --- | --- |
| 5 | Herramientas, framework y multijugador |

# Inicio

Partiremos de una versión muy simplificada del juego plataformas, la cual **descargaremos del campus virtual**.

Cargamos el proyecto y probamos que funciona correctamente, se dispara con el enter.

# Cache

Comenzaremos incluyendo un sistema de cache más eficiente en el fichero **Res.js**. Además de crear las imágenes las guardaremos en una tabla Hash (Map).

|  |
| --- |
| **var *cache*** = []; **var *imagenes*** = {  **prueba\_tiles32** : **"res/prueba\_tiles32.png"**,  **caja\_madera** : **"res/caja\_madera.png"**, |

En la función la **cargarImagenes** vamos a guardar las imágenes cargadas en la tabla hash **cache**, como clave utilizaremos la propia ruta de cada fichero.

|  |
| --- |
| **var *rutasImagenes*** = ***Object***.values(***imagenes***); *cargarImagenes*(0);  **function** *cargarImagenes*(indice){ **~~var~~** ~~imagenCargar =~~ **~~new~~ *~~Image~~***~~();  imagenCargar.src =~~ ***~~rutasImagenes~~***~~[indice];~~  ***cache***[***rutasImagenes***[indice]] = **new *Image***();  ***cache***[***rutasImagenes***[indice]].**src** = ***rutasImagenes***[indice];  ***cache***[***rutasImagenes***[indice]].onload = **function**(){  **if** ( indice < ***rutasImagenes***.**length**-1 ){  indice++;  *cargarImagenes*(indice);  } **else** {  *iniciarJuego*();  }  } } |

**Nueva alternativa para cargar fotos:** ahora tenemos la posibilidad adicional de acceder directamente a la imagen a través de la variable **cache[ruta]**.

Por ejemplo, en el constructor de **Modelo.js** podemos referenciar la imagen que hay en cache en lugar de crear una nueva.

|  |
| --- |
| constructor(imagenRuta, x, y) {  **~~this~~**~~.~~**~~imagen~~** ~~=~~ **~~new~~ *~~Image~~***~~();~~ **~~this~~**~~.~~**~~imagen~~**~~.~~**~~src~~** ~~= imagenRuta;~~  **this**.**imagen** = ***cache***[imagenRuta];  **this**.**x** = x;  **this**.**y** = y;  **this**.**ancho** = **this**.**imagen**.**width**; |

|  |  |
| --- | --- |
| El sistema de **cache** se puede aplicar en todos los sitios en los que se usan imágenes, por ejemplo, en el constructor de **Animacion.js**   |  | | --- | | **class** Animacion {   constructor(imagenSrc, modeloAncho, modeloAlto, velocidadRefresco, framesTotales, callback) {  *// Nuevo para animaciones finitas* **this**.**callback** = callback;    **~~this~~**~~.~~**~~imagen~~** ~~=~~ **~~new~~ *~~Image~~***~~();~~ **~~this~~**~~.~~**~~imagen~~**~~.~~**~~src~~** ~~= imagenSrc;~~  **this**.**imagen** = ***cache***[imagenSrc]; | |

Ejecutamos el juego y probamos que todo funciona como antes (no tiene cambios visibles)



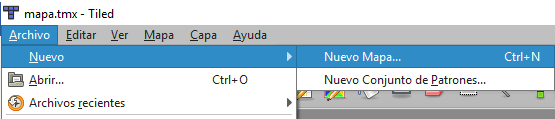
|  |  |
| --- | --- |
| Atención, si el proyecto en el que hemos incluido la cache tiene el fondo móvil (nubes que se mueven, **parte opcional de la práctica 2** ), el sistema de cache nos puede dar un error al crear la foto secundaria.   |  | | --- | | **class** Fondo **extends** Modelo {   constructor(rutaImagen, x, y) {  **super**(rutaImagen, x, y)  **this**.**vx** = 0;  **this**.rutaImagen= rutaImagen;  }   actualizar(){  **if** ( **this**.**vx** != 0) {  **if** ( **this**.**fondoAux** == **null**){  **this**.**fondoAux** =  **new** Fondo(rutaImagen, **this**.**x**, **this**.**y**);  } | |

# Mapa con Tiled

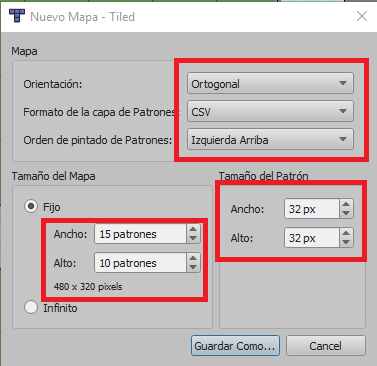
Utilizaremos el editor Tiled, para definir un mapa de forma gráfica.

Descargamos el editor de mapas de la página oficial <https://www.mapeditor.org/news.html> **Utilizaremos la versión 1.2.4** (puede haber pequeños cambios entre versiones).

Abrimos el editor, si no nos aparece un nuevo mapa creado, lo creamos **Nuevo -> Nuevo Mapa**

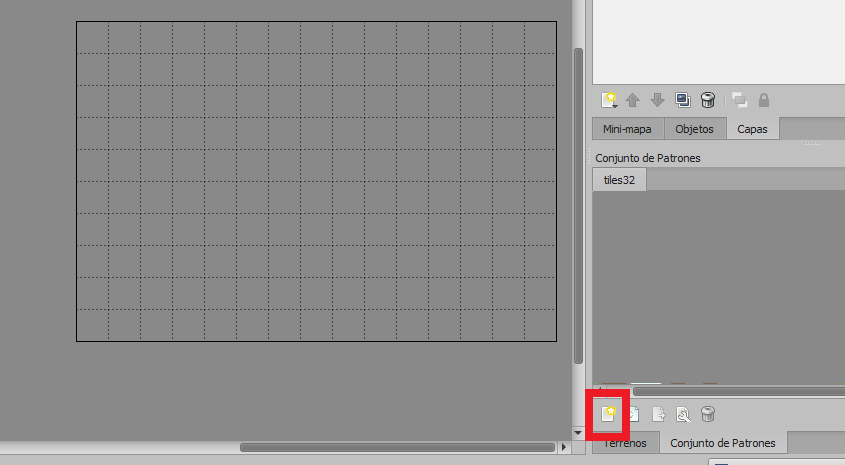


Nos fijamos en que todas las opciones **coincidan exactamente** con la siguiente captura (Orientación, Formato de la capa de Patrones, Orden de pintado de Patrones, etc.)

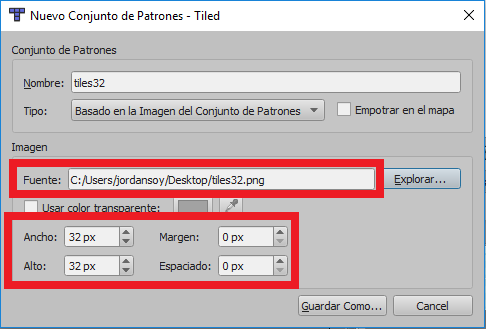


Sí nos solicita guardar el mapa lo hacemos con el nombre **mapa.tmx**

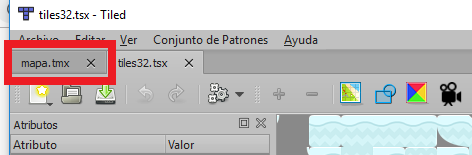
Lo primero que vamos a hacer es agregar una capa de patrones.



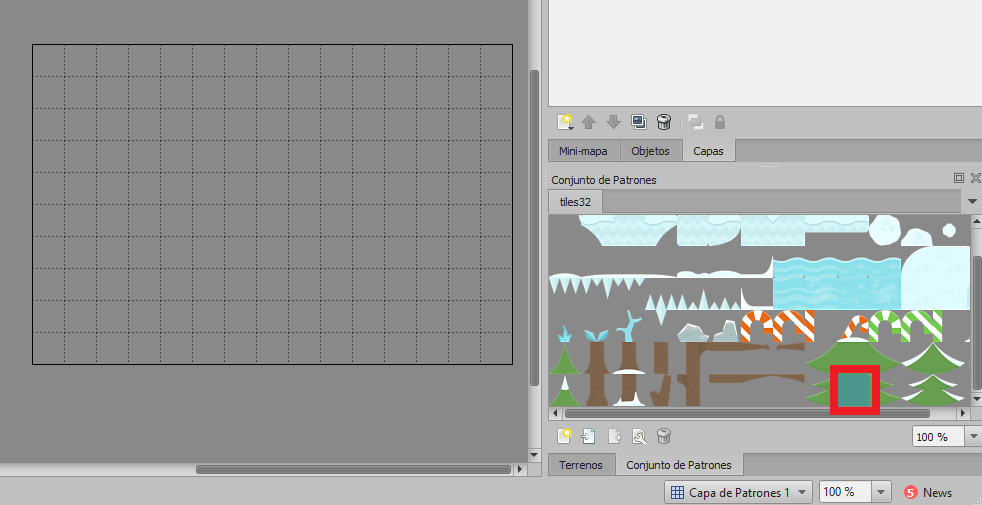
Seleccionamos el fichero **tiles32.png** (está en los recursos del campus virtual) y nos aseguramos de que el ancho y el alto sea correcto (los tiles que vamos a usar son de **32** x **32**).



Pulsamos en **guardar como**, si se nos cambia de vista volvemos a acceder a la pestaña del **mapa**.**tmx**.

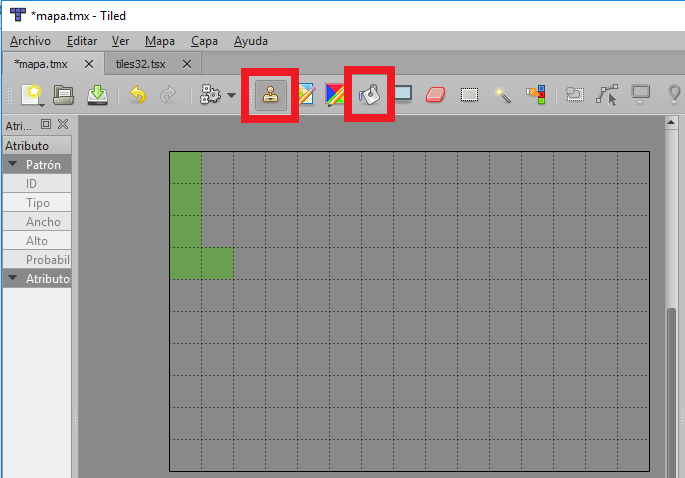


Para colocar un tile sobre el mapa lo primero es seleccionarlo.

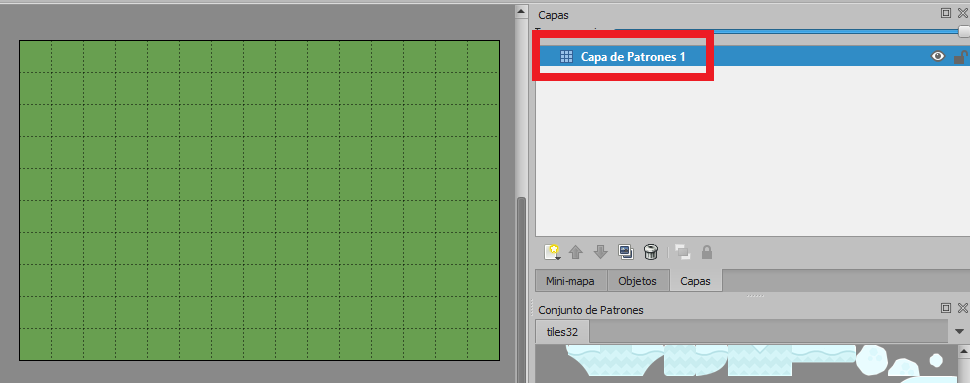


Hay dos herramientas principales para colocar Tiles.

* Herramienta de estampar, coloca tiles de uno en uno
* Herramienta de rellanar, rellena superficie dentro de un límite

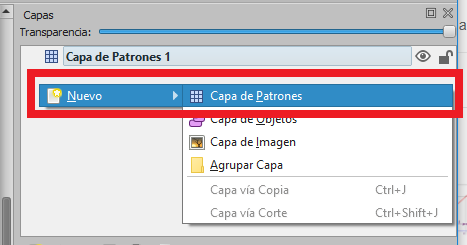


Cubrimos toda la capa de verde

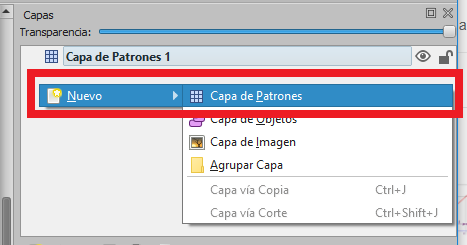


Todos estos tiles los estamos colocando en la **Capa de patrones 1**.

Estos mapas se caracterizan por que pueden tener múltiples capas. Vamos a crear una adicional, **Click derecho -> Nuevo -> Capa de Patrones**.

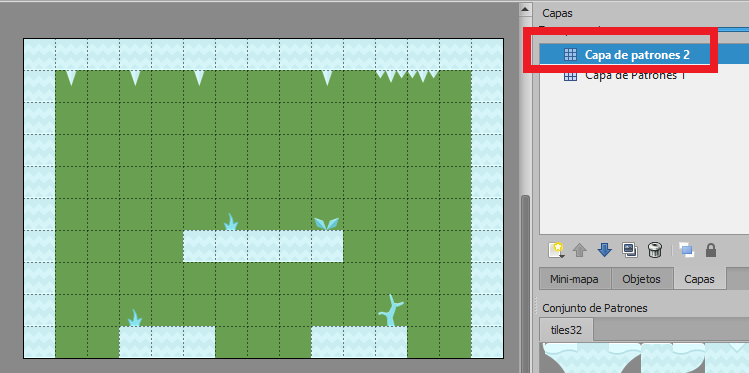


Seleccionamos la nueva capa, de tipo **Capa de Patrones.**



La capa 2 está literalmente encima de la capa 1.

Seleccionamos otros tiles y decoramos la segunda capa (hay que asegurarse de tener la **Capa de patrones 2** seleccionada).

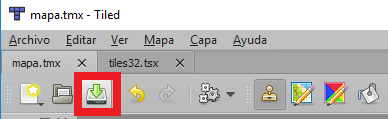


En la captura podemos ver como las capas están **superpuestas**, primero se pintan los elementos de la capa 1 y la capa 2.

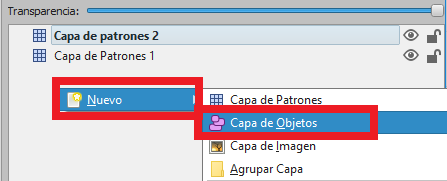


|  |
| --- |
| Se pueden incluir tantas capas de patrones como se quiera, suelen depender de la complejidad del mapa a modelar y de la necesidad de colocar elementos superpuestos, por ejemplo, en los juegos RPG se suele usar un mínimo de 3:   * Capa 1: terrenos, hierba, arena, etc. * Capa 2: Muros de casas y arboles * Capa 3: Decoración sobre casas: tejados, ventanas, puertas, etc.   Ver el ejemplo del mapa en el campus virtual. |

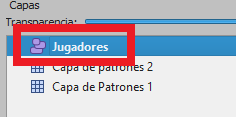
**\*Importante:** Tiled, no tiene auto-salvado, así que debemos guardar el mapa cada poco.



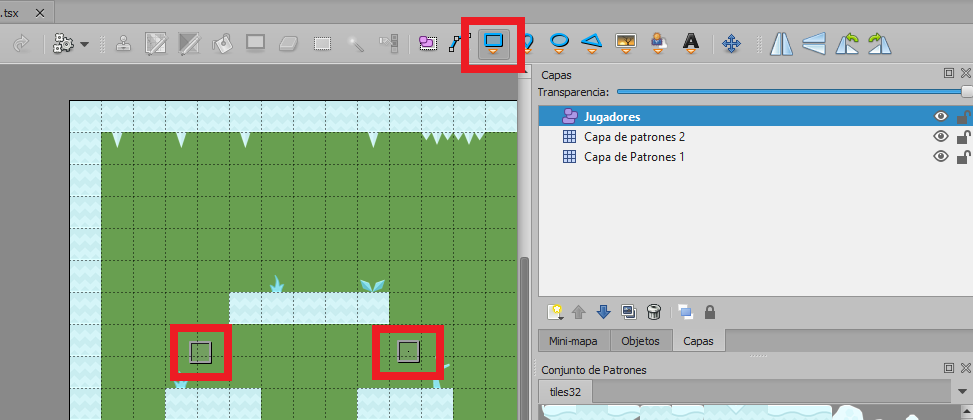
Para colocar al Jugador (enemigos, etc.) debemos colocar **capas de objetos.** Se usa una **capa de objetos** por cada elemento que se quiera colocar.



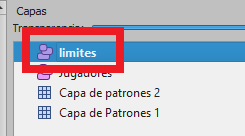
Es muy recomendable, cambiarle el nombre a la capa, por ejemplo, **Jugadores**.



Una vez creada la capa, hay que seleccionarla y colocar un objeto encima. (El objeto puede ser de cualquier tipo pues solo nos interesa sus coordenadas x e y). Colocamos 2 jugadores.



Nos vamos a adelantar, y vamos a crear una nueva capa de objetos que llamaremos **limites**.

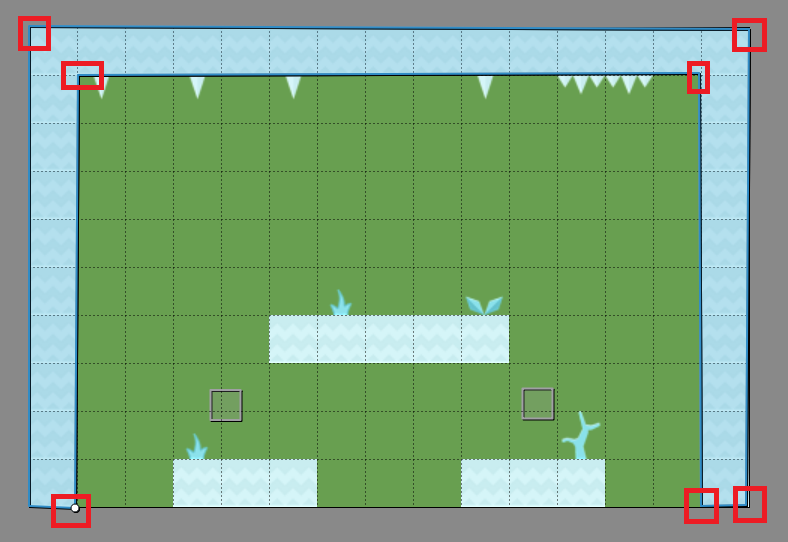


Los limites van a ser lo que suponga las colisiones reales en el juego, los tiles que hemos colocado es **pura decoración** (similar a un fondo creado a partir de imágenes más pequeñas).

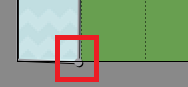
Seleccionamos la herramienta **polilínea** (sucesión de puntos para crear un polígono).

Vamos colocando líneas en las partes que queramos que haya colisiones.

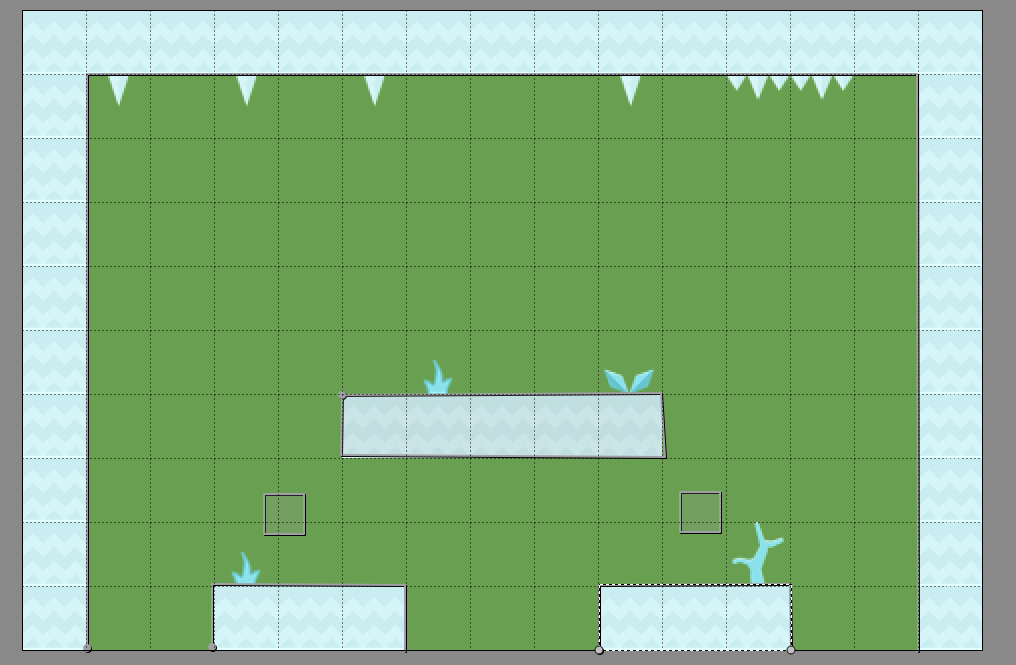
* **Click derecho:** colocar punto
* **Click izquierdo:** acabar línea, a partir de ahí se puede empezar una nueva.



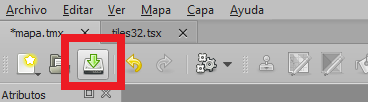
**Cerrar siempre el polígono! (el último punto debe unirse con el primero)**



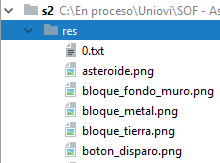
Debería quedar algo así (hemos hecho que la decoración no sea “colisionable”).



Nos aseguramos de guardar todos los cambios.



Localizamos el fichero **mapa.tmx** en nuestro equipo y lo copiamos dentro de la carpeta **/res** del juego.



Podemos abrirlo para ver su formato, es un **fichero xml**



Observamos que el XML define todos los elementos que incluimos de forma gráfica.

Copiamos en la carpeta **/res** de nuestro proyecto el fichero **tiles32.png**, lo agregamos a Res.js

|  |
| --- |
| **var *cache*** = [] **var *imagenes*** = {  **tiles32** : **"res/tiles32.png"**,  **prueba\_tiles32** : **"res/prueba\_tiles32.png"**,  **caja\_madera** : **"res/caja\_madera.png"**, |

# Cargar el mapa

Abrimos el **GameLayer.js**, en su método **iniciar(),** cambiamos la llamada final

|  |
| --- |
| **this**.**disparosJugador** = []  **this**.**puntos** = **new** Texto(0,480\*0.9,320\*0.07 );  **~~this~~**~~.cargarMapa(~~**~~"res/"~~**~~+~~***~~nivelActual~~***~~+~~**~~".txt"~~**~~);~~  **this**.cargarMapaTmx(**"res/mapa.tmx"**); } |

Creamos una nueva función **cargarMapaTmx.** Vamos a empezar por lo más sencillo, cargar al Jugador.

|  |
| --- |
| cargarMapaTmx(ruta){  **var** fichero = **new *XMLHttpRequest***();  fichero.open(**"GET"**, ruta, **false**);   fichero.onreadystatechange = **function** () {  **var** texto = fichero.responseText;  **var** parser = **new *DOMParser***();  **var** xmlDoc = parser.parseFromString(texto,**"text/xml"**);   **var** grupoJugadores = xmlDoc.getElementsByTagName(**"objectgroup"**)[0];  *//* ***Todo lo que se lee del XML se parsea a int* var** x = grupoJugadores.children[0].getAttribute(**"x"**);  x = *parseInt*(x);  y = *parseInt*(y);  **var** y = grupoJugadores.children[0].getAttribute(**"y"**);  **this**.cargarObjetoMapa(**"1"**,x,y);   **var** x = grupoJugadores.children[1].getAttribute(**"x"**);  **var** y = grupoJugadores.children[1].getAttribute(**"y"**);  x = *parseInt*(x);  y = *parseInt*(y);  **this**.cargarObjetoMapa(**"2"**,x,y);   }.bind(**this**);   fichero.send(**null**); } |

Aun no estamos cargando el Jugador 2.

|  |
| --- |
| Navegación por XML – **nodos y atributos**  El objeto **xmlDoc**, nos permitirá navegar por el XML y obtener la información deseada en cada caso. Los XML definen información con el siguiente formato:    En un xml podemos encontrar elementos/nodos, con una etiqueta de apertura y cierre, atributos y contenidos.  **xmlDoc.getElementsByTagName(** <nombre> **)** retorna un array con los elementos XML que tengan el nombre indicado, retorna una lista porque pueden ser varios.    **Children[indice]** permite colocarse sobre un hijo concreto de un elemento, por ejemplo.    **getAttribute(<atributo>)** nos permite consultar el valor de un atributo de un elemento, debemos tener actualmente la referencia al elemento, por ejemplo: |

|  |
| --- |
| **Importante** – La lectura de datos de un fichero siempre se hace de forma textual, por eso es importante utilizar **parseInt(valor)**  De lo contrario tendremos “1” en lugar de 1 y al hacer una operación + estaremos haciendo una concatenación. |

Sí probamos el juego aparecerá un jugador que caerá **(es posible que haya que limpiar la cache del navegador)**



Lo siguiente que haremos será generar un fondo nuevo a partir de los Tiles. Para ello vamos a crear un array de **fondos** en el **iniciar()** del **GameLayer.js**

|  |
| --- |
| iniciar() {  **this**.**espacio** = **new** Espacio(1);   **this**.**scrollX** = 0;  **this**.**bloques** = [];   **this**.**fondo** = **new** Fondo(***imagenes***.**fondo\_2**,480\*0.5,320\*0.5);  **this**.**fondos** = []; |

En el **dibujar()**, debemos dibujar los fondos, lo hacemos a continuación del fondo principal.

Aprovechamos para eliminar la llamada a **this.calcularSroll(),** pues no queremos que este juego lo tenga.

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

Para dibujar fondos a partir de tiles tenemos que modificar la clase **Fondo.js** en primer lugar ampliando el número de parámetros que pueden ser recibidos por su constructor.

Sí nos fijamos en el fichero **mapa.tmx** los tiles vienen dados por un índice, por ejemplo 3, 34, 52, 53, etc. (Es la posición que ocupan en la foto tiles32)



Ilustración 1 Tiles32

La clase **fondo** debe poder recortar el cuadrado exacto de tiles32 y añadirlo al Canvas.

El parámetro opcional **i** va a representar el índice del tile que va a pintar, como la foto tiles32 se divide en filas y columnas habrá que calcular con que fila y columna se corresponde i.

|  |
| --- |
| **class** Fondo **extends** Modelo {   constructor(rutaImagen, x, y, i, columnas, filas) {  **super**(rutaImagen, x, y)   **if** (i != **null**) {  **this**.**i** = i;  **this**.**fila** = *parseInt*(i / columnas);  **if** ( **this**.**fila** > 0){  **this**.**columna** = **this**.**i** - (**this**.**fila** \* columnas);  } **else** {  **this**.**columna** = **this**.**i**;  }  }  } |

Ahora debemos modificar el **dibujar()**, en caso de que “i” tenga valor, la imagen será el elemento recortado de Tiles32 correspondiente.

Cabe destacar que sí i == 0 no queremos que se pinte nada. 0 significa que no hay tile.

|  |
| --- |
| dibujar (){  **if** (**this**.i != **null**) {  **if** ( **this**.i != 0) {  ***contexto***.drawImage(  **this**.**imagen**,   **this**.**columna** \* 32 - 32, **this**.**fila** \* 32 ,  32, 32,  **this**.**x** - 32/2, **this**.**y** - 32/2,  32, 32);  }  } **else** {  ***contexto***.drawImage(**this**.**imagen**,  **this**.**x** - **this**.**imagen**.**width**/2 - ***scrollX***,  **this**.**y** - **this**.**imagen**.**height**/2);  } } |

Volvemos al **GameLayer.js** a la función **cargarMapaTmx(),** empezamos a leer las capas de tiles y a crear los fondos correspondientes.

|  |
| --- |
| **var** x = grupoJugadores.children[1].getAttribute(**"x"**);  **var** y = grupoJugadores.children[1].getAttribute(**"y"**);  x = *parseInt*(x);  y = *parseInt*(y);  **this**.cargarObjetoMapa(**"2"**,*parseInt*(x),*parseInt*(y));     *// Capas de tiles* **var** capasTiles = xmlDoc.getElementsByTagName(**"layer"**);  **for**(**var** t= 0; t < capasTiles.length; t++) {  **var** contenido = capasTiles[t].children[0].childNodes[0].nodeValue;  **var** fondos = contenido.split(**'\n'**);  fondos.splice(0, 1); *// la primera está vacía* fondos.splice(fondos.length - 1, 1); *// la final está vacía* **for** (**var** i = 0; i < fondos.length; i++) {  **var** linea = fondos[i].split(**','**);  **for** (**var** j = 0; j < linea.length; j++) {  **var** tile = **new** Fondo(***imagenes***.**tiles32**,  32 / 2 + j \* 32,  32 / 2 + i \* 32,  *parseInt*(linea[j]),  14,  7  ); *// 14 columnas, 7 filas* **this**.**fondos**.push(tile);  }  }  }  }.bind(**this**);  fichero.send(**null**); |

|  |
| --- |
| Lector de XML – contenido de los nodos  **childNodes[0].NodeValue** – permite leer el contenido de un nodo XML previamente seleccionado. El contenido de la etiqueta XML es considerado como un hijo adicional, por eso debemos hacer un childNode para acceder hasta él antes de leer su contenido |

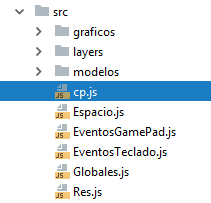
Ejecutamos el juego y comprobamos que se dibuja un fondo a partir de los tiles.

****

# Motor de fuerzas físicas - Chipmunk

En esta parte del guion vamos a integrar el motor de fuerzas físicas 2D <https://chipmunk-physics.net/> en nuestro proyecto.

Copiamos la clase **cp.js** dentro de **/src/** (descargarnos el fichero del campus virtual).



Agregamos el fichero al **index.html**

|  |
| --- |
| <**script src="src/cp.js"**></**script**> <**script src="src/EventosGamePad.js"**></**script**> <**script src="src/Espacio.js"**></**script**> <**script src="src/Globales.js"**></**script**> <**script src="src/Eventosteclado.js"**></**script**> <**script src="src/Res.js"**></**script**> <**script src="Main.js"**></**script**> |

En la función **iniciar()** del **GameLayer.js** instanciamos el nuevo espacio (podemos eliminar la referencia al espacio anterior).

La gravedad se define mediante el vector **cp.v(0,0.0006)**, los vectores cp.v solo tienen una coordenada, la coordenada “inicial” de vector se supone la 0.0 por lo tanto el vector va desde la 0.0 a la 0.0006 .

|  |
| --- |
| iniciar() {  **~~this~~**~~.~~**~~espacio~~** ~~=~~ **~~new~~** ~~Espacio(1);~~    **this**.**space** = **new cp**.Space();  **this**.**space**.**gravity** = **cp**.v(0, 0.0006);    **this**.**scrollX** = 0; |

Debemos modificar la función **actualizar(),** para que utilice el nuevo espacio el equivalente a la actualización del espacio en Chipmunk es la función **step(ms\_transcurridos)**

|  |
| --- |
| actualizar (){  **~~this~~**~~.~~**~~espacio~~**~~.actualizar();~~  **this**.**space**.step(1000/30); |

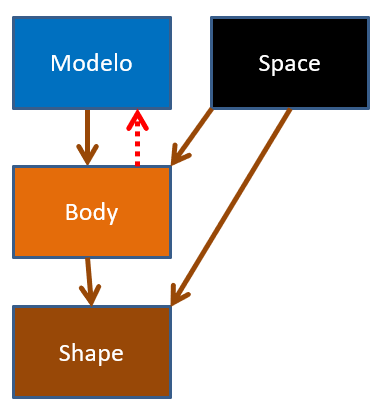
El primer modelo que utilizará el nuevo espacio será el **Jugador.js**. El espacio solo gestiona datos relacionados con las físicas, nada de gráficos.

|  |
| --- |
| No es nuevo que el espacio gestione datos relacionados con las físicas, esto también ocurría en el espacio que nosotros habíamos implementado. La diferencia radica en que nuestro espacio se integraba perfectamente con modelo y este en cambio tendremos que integrarlo manualmente.  El mayor reto de la integración es pasar las coordenadas X e Y del espacio al modelo en cada frame del juego. |

Elementos.

* El **body (cuerpo)** define la masa y la posición del elemento
* La **shape (forma),** está asociada al body y define la forma (por ejemplo, un cuadrado), la elasticidad y la fricción. La shape será importante para las colisiones

La relación entre los elementos es la siguiente.

****

Creamos la función **configurarEspacio()** en el **Jugador.**

|  |
| --- |
| configurarEspacio(space){   *//1 Crear cuerpo* **this**.**body** = **new cp**.Body(3, ***Infinity***);  space.addBody(**this**.**body**);   *//posicionar cuerpo (El cuerpo tiene la posición)* **this**.**body**.**p**.**x** = **this**.**x**;  **this**.**body**.**p**.**y** = **this**.**y**;   *//2 Crear forma (asociada al body)* **this**.**shape** = **new cp**.BoxShape(**this**.**body**,  **this**.**ancho**, **this**.**alto**);  **this**.**shape**.setFriction(0.8);  **this**.**shape**.setElasticity(0);  space.addShape(**this**.**shape**);   *//3 Extra - Referencia cruzada, que el cuerpo acceda al modelo  // (Simplifica cosas)* **this**.**body**.**modelo** = **this**; } |

**Importante**, el espacio de Chipmunk solamente modifica las posiciones X e Y del cuerpo **body (body.p.x , body.p.y)** , no las del modelo, para que se pinte en el mismo sitio que en su espacio físico debemos hacer que las posiciones se copien al modelo.

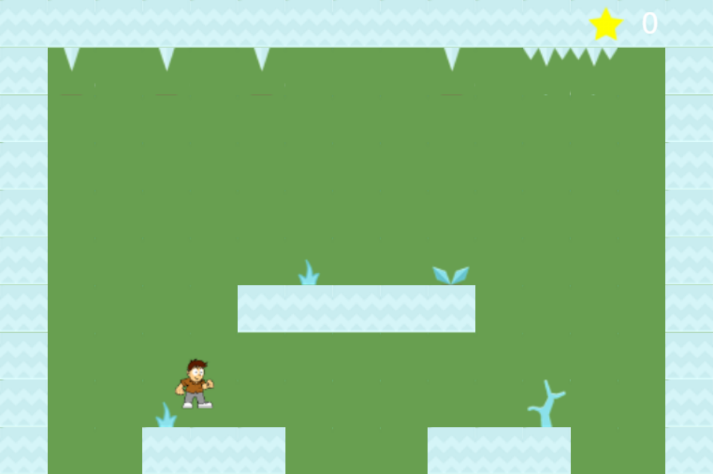
Modificamos el método **actualizar()** del **Jugador**, aprovechamos también para dejar de usar la variable **this.vx**, y en su lugar usar **this.body.vx** (velocidad real utilizada en el espacio Chipmunk).

|  |
| --- |
| actualizar(){  **this**.**x** = **this**.**body**.**p**.**x**;  **this**.**y** = **this**.**body**.**p**.**y**;    **this**.**animacion**.actualizar();   *// ¿Esta en el aire?* **if** (**this**.**choqueAbajo** == **true**){  **this**.**enElAire** = **false**;  } **else** {  **this**.**enElAire** = **true**;  }   *// Establecer orientación* **if** ( **this**.**body**.**vx** > 0.01 ){  **this**.**orientacion** = ***orientaciones***.**derecha**;  }  **if** ( **this**.**body**.**vx** < -0.01 ){  **this**.**orientacion** = ***orientaciones***.**izquierda**;  }   **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**.**body**.**vx** > 0.01 || **this**.**body**.**vx** < -0.01) {  **if** (**this**.**orientacion** == ***orientaciones***.**derecha**) {  **this**.**animacion** = **this**.**aCorriendoDerecha**;  }  **if** (**this**.**orientacion** == ***orientaciones***.**izquierda**) {  **this**.**animacion** = **this**.**aCorriendoIzquierda**;  }  }  **if** ( **this**.**body**.**vx <** 0.01 && **this**.**body**.**vx >** -0.01){  **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**--;  } } |

Volvemos al **GameLayer.js**, en la función **cargarObjetoMapa(),** debemos llamar a la función que configura las físicas del jugador (y eliminar la versión anterior).

|  |
| --- |
| 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;  **this**.**jugador**.configurarEspacio(**this**.**space**);  **~~this~~**~~.~~**~~espacio~~**~~.agregarCuerpoDinamico(~~**~~this~~**~~.~~**~~jugador~~**~~);~~  **break**; |

Probamos el juego, el jugador debería caer (por el momento es el único elemento del espacio físico, no hemos definido suelos ni nada similar)

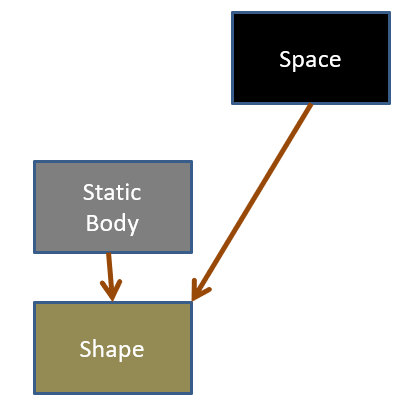


Vamos a agregar los límites que hemos dibujado en el mapa como restricciones al espacio físico

Para ello ampliamos el **cargarMapaTmx().**

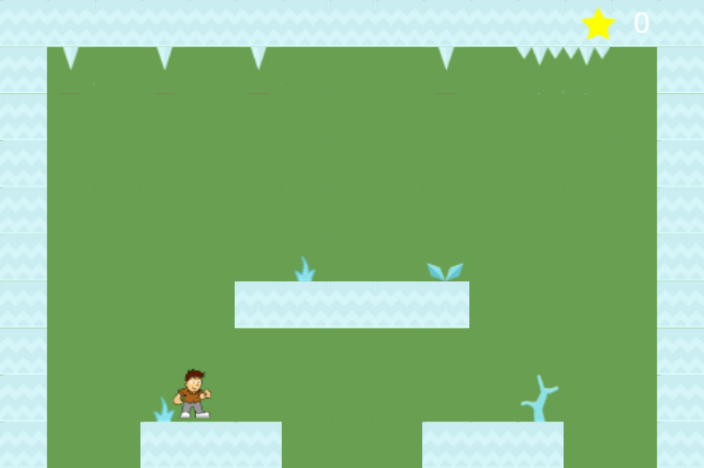
|  |
| --- |
| Procesaremos la lista de puntos de cada polígono definido en el XML.    Para cada par de puntos vamos a crear un **body (estático)** y un shape de tipo especial llamado **SegmentShape**, es básicamente una línea que recibe dos puntos (inicio y fin de la línea)  Estas líneas ni siquiera van a tener modelo gráfico van a ser solo restricciones en el espacio.  **Este código es reutilizable para cualquier sistema de limites basado en polígonos.** |

La definición de elementos estáticos en el espacio es notablemente más simple que la de dinámicos. No hace falta agregar el body al espacio (ya que realmente nunca se mueve, su masa no importa). La forma se agrega al espacio con la función addStaticShape en lugar de addShape.



|  |
| --- |
| }  }   *// Limites* **var** lim = xmlDoc.getElementsByTagName(**"objectgroup"**)[1].children;  **for** (**var** i=0; i < lim.length; i++ ){  **var** linea = lim[i].children[0].getAttribute(**"points"**);  **var** p = linea.split(**' '**);**for**(**var** j = 0; j < p.length - 1; j++) {  **var** x = *parseInt*(lim[i].getAttribute(**"x"**));  **var** y = *parseInt*(lim[i].getAttribute(**"y"**));  **var** px1 = *parseInt*(p[j].split(**','**)[0]);  **var** py1 = *parseInt*(p[j].split(**','**)[1]);  **var** px2 = *parseInt*(p[j+1].split(**','**)[0]);  **var** py2 = *parseInt*(p[j+1].split(**','**)[1]);   *//1 Crear cuerpo Estatico* **var** bodySuelo = **this**.**space**.**staticBody**;  *// space.addBody(bodySuelo); NO HACE FALTA AGREGAR LOS ESTATICOS   // 2 Crear forma (asociada al body)* **var** shapeSuelo = **new cp**.SegmentShape(bodySuelo,  **cp**.v(x + px1, y + py1),  **cp**.v(x + px2, y + py2), 1);   shapeSuelo.setFriction(1);  shapeSuelo.setElasticity(0.8);  **this**.**space**.addStaticShape(shapeSuelo);  }  }  }.bind(**this**); |

Probamos que aparecen colisiones con las líneas estáticas.



# Mover

Modificamos la función **moverX(dirección)**, eliminamos el uso de la variables **this.vx** (obsoleta, aunque no daría errores mantenerla). Utilizamos el sistema de impulsos para moverlo hacia la derecha y la izquierda, comprobando que los impulsos solo se añaden cuando la velocidad está dentro de unos límites ( 0.15 y -0-15)

Los impulsos se aplican directamente sobre un **body,** debemos utilizar impulsos pequeños para no corromper el motor de físicas.

**Body.applyImpulse** permite definir un impulso, el primer valor que recibe es el más importante el vector con la fuerza que se desea aplicar.

|  |
| --- |
| moverX (direccion){  **~~this~~**~~.~~**~~vx~~** ~~= direccion \* 3;~~    **if** (direccion > 0 && **this**.**body**.**vx** < 0.15) {  **this**.**body**.applyImpulse(**cp**.v(0.15, 0.08), **cp**.v(0, 0));  }   **if** (direccion < 0 && **this**.**body**.**vx** > -0.15) {  **this**.**body**.applyImpulse(**cp**.v(-0.15, 0.08), **cp**.v(0, 0));  } } |

Para incluir el salto vamos a agregar un contador de instantes en los que el Jugador permanece con una velocidad **vy** muy próxima a 0 (es casi imposible tener 0 ya que la variable **vy** es flotante, siempre se tendrá velocidad 0,001 o similar)

|  |
| --- |
| constructor(x, y) {  **super**(***imagenes***.**jugador** , x, y)  **this**.**quieto**=0;  **this**.**estado** = ***estados***.**moviendo**; |

Dentro del **actualizar()**, incluimos el sumador de la variable **this.quieto**, estableceremos que el jugador está parado cuando tiene una **vy** entre 0.01 y -0.01

**Importante**, eliminamos el antiguo sistema de choque que determinaba cuando el jugador estaba en el aire.

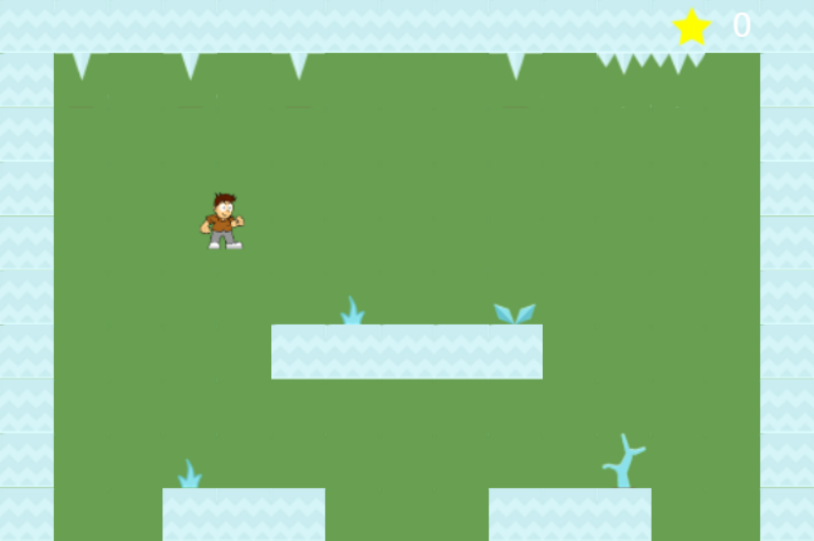
|  |
| --- |
| actualizar(){  **this**.**x** = **this**.**body**.**p**.**x**;  **this**.**y** = **this**.**body**.**p**.**y**;   **this**.**animacion**.actualizar();    **if** (**this**.**body**.**vy** < 0.01 && **this**.**body**.**vy** > -0.01 ){  **this**.**quieto**++;  **if** (**this**.**quieto** >= 3){  **this**.**enElAire** = **false**;  }  } **else** {  **this**.**quieto** = 0;  **this**.**enElAire** = **true** }   *~~// ¿Esta en el aire?~~* **~~if~~** ~~(~~**~~this~~**~~.~~**~~choqueAbajo~~** ~~==~~ **~~true~~**~~){~~ **~~this~~**~~.~~**~~enElAire~~** ~~=~~ **~~false~~**~~;  }~~ **~~else~~** ~~{~~ **~~this~~**~~.~~**~~enElAire~~** ~~=~~ **~~true~~**~~;  }~~ |

Una vez se acumulen 3 iteraciones sin “demasiada” velocidad **vy** estableceremos que no está en el aire, es decir, puede saltar.

Modificamos la función **saltar().**

|  |
| --- |
| saltar(){  **if** ( !**this**.**enElAire** ) {  **~~this~~**~~.~~**~~vy~~** ~~= -16;~~  **this**.**body**.applyImpulse(**cp**.v(0, -1.2), **cp**.v(0, 0));  **this**.**enElAire** = **true**;  } } |

Probamos el sistema de movimiento.



# Disparo

Falta introducir la clase **DisparoJugador** en el espacio de Chipmunk.

Abrimos **DisparoJugador.js** y agregamos la función **configurarEspacio** (va a ser casi igual a la configuración del jugador).

|  |
| --- |
| configurarEspacio(space) {  *//1 Crear cuerpo* **this**.**body** = **new cp**.Body(1, ***Infinity***);  space.addBody(**this**.**body**);   *//posicionar cuerpo (El cuerpo tiene la posición)* **this**.**body**.**p**.**x** = **this**.**x**;  **this**.**body**.**p**.**y** = **this**.**y**;   *//2 Crear forma (asociada al body)* **this**.**shape** = **new cp**.BoxShape(**this**.**body**,  **this**.**ancho**, **this**.**alto**);  **this**.**shape**.setElasticity(0.6);  space.addShape(**this**.**shape**);   *//3 Extra - Referencia cruzada, que el cuerpo acceda al modelo  // (Simplifica cosas)* **this**.**body**.**modelo** = **this**; } |

**Importante** - **Recordar**. Al igual que hicimos con el Jugador, debemos pasar las coordenadas del **body** al **modelo** en cada iteración del **actualizar()** (sino no se pintará en la coordenada que tiene en el espacio).

|  |
| --- |
| actualizar (){  **this**.**x** = **this**.**body**.**p**.**x**;  **this**.**y** = **this**.**body**.**p**.**y**; } |

Ahora debemos a acceder a la parte donde se crean los disparos y se agregan al espacio.

Abrimos el **GameLayer** y accedemos a **procesarControles()** y eliminamos la referencia a agregar el disparo al antiguo espacio, en lugar de ello llamamos al **configurarEspacio()** que implementamos en el Disparo.

Como hemos visto anteriormente, el movimiento a los cuerpos se aplica mediante impulsos. Aplicamos un impulso en la coordenada X positivo (0.35) o negativo (-0.35) dependiendo de la orientación del jugador, la coordenada Y la dejamos siempre negativa (-0.1) para que el disparo salga hacia arriba.

En lugar de hacer que los disparos aparezcan en la misma coordenada x del jugador, los movemos 25 o -25 pixeles (aparecerán un poco delante).

|  |
| --- |
| procesarControles( ){  *// disparar* **if** ( ***controles***.**disparo** ){  **var** nuevoDisparo = **this**.**jugador**.disparar();  **if** ( nuevoDisparo != **null** ) {  **t~~his~~**~~.espacio.agregarCuerpoDinamico~~(nuevoDisparo);  nuevoDisparo.configurarEspacio(**this**.**space**);  **if** ( **this**.**jugador**.**orientacion** == ***orientaciones***.**izquierda** ){  nuevoDisparo.body.applyImpulse(**cp**.v(-0.35, -0.1), **cp**.v(0, 0));  nuevoDisparo.body.p.**x** = nuevoDisparo.body.p.**x** - 25;  } **else** {  nuevoDisparo.body.applyImpulse(**cp**.v(0.35, -0.1), **cp**.v(0, 0));  nuevoDisparo.body.p.**x** = nuevoDisparo.body.p.**x** + 25;  }  **this**.**disparosJugador**.push(nuevoDisparo);   }  } |

Probamos a ejecutar el juego.



# Segundo Jugador

Comenzamos implementando el sistema de control para el segundo jugador. Abrimos **Main.js** y creamos la variable **controles2**.

|  |
| --- |
| *Ç// Canvas y contexto del Canvas* **var *canvas*** = ***document***.getElementById(**"canvas"**); **var *contexto*** = ***canvas***.getContext(**"2d"**); **var *escaladoMinimo*** = 1;  *// Controles* **var *controles*** = {}; **var *controles2*** = {}; |

Modificamos el **EventosTeclado.js** para que utilice el **controles2**. El nuevo jugador se controlará con W,A,S,D y SPACE. Comenzamos incluyendo los casos relativos al **onKeyDown().**

|  |
| --- |
| **function** *onKeyDown*( event) {  ***entrada*** = ***entradas***.**teclado**;  *// agregar la tecla pulsada si no estaba* ***console***.lo  **var** posicion = ***teclas***.indexOf(event.keyCode)  *alert*(event.keyCode);  **if** ( posicion == -1 ) {  ***teclas***.push(event.keyCode);  **switch** ( event.keyCode ){  **case** 32:  ***controles2***.**continuar** = **true**;  ***controles2***.**disparo** = **true**;  **break**;  **case** 87:  ***controles2***.**moverY** = 1;  **break**;  **case** 83:  ***controles2***.**moverY** = -1;  **break**;  **case** 68:  ***controles2***.**moverX** = 1;  **break**;  **case** 65:  ***controles2***.**moverX** = -1;  **break**;  **case** 13:  ***controles***.**continuar** = **true**;  ***controles***.**disparo** = **true**;  **break**;  **case** 38:  ***controles***.**moverY** = 1;  **break**;  **case** 40: |

Incluimos los casos del **onKeyUp().**

|  |
| --- |
| **function** *onKeyUp*( event) {  *// sacar la tecla pulsada* **var** posicion = ***teclas***.indexOf(event.keyCode);  ***teclas***.splice( posicion, 1);  **switch** ( event.keyCode ){  **case** 32:  ***controles2***.**continuar** = **false**;  ***controles2***.**disparo** = **false**;  **break**;  **case** 87:  **if** ( ***controles2***.**moverY** == 1 ){  ***controles2***.**moverY** = 0;  }  **break**;  **case** 83:  **if** ( ***controles2***.**moverY** == -1 ){  ***controles2***.**moverY** = 0;  }  **break**;  **case** 68:  **if** ( ***controles2***.**moverX** == 1 ){  ***controles2***.**moverX** = 0;  }  **break**;  **case** 65:  **if** ( ***controles2***.**moverX** == -1 ){  ***controles2***.**moverX** = 0;  }  **break**;  **case** 13:  ***controles***.**continuar** = **false**;  ***controles***.**disparo** = **false**;  **break**; |

Accedemos al **GameLayer.js** y procedemos a cargar el jugador en la función **cargarObjetoMapa()**, determinamos el carácter “2” para crear al jugador 2, al cual vamos a llamar **contrario**.

|  |
| --- |
| cargarObjetoMapa(simbolo, x, y){  **switch**(simbolo) {  **case "2"**:  **this**.**contrario** = **new** Jugador(x, y);  *// modificación para empezar a contar desde el suelo* **this**.**contrario**.**y** = **this**.**contrario**.**y** - **this**.**contrario**.**alto**/2;  **this**.**contrario**.configurarEspacio(**this**.**space**);   **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**.**jugador**.configurarEspacio(**this**.**space**); |

Nos aseguramos de actualizar al **contrario**, en cada iteración del **actualizar()**.

|  |
| --- |
| actualizar (){  **this**.**space**.step(1000/30);   *// actualizar* **this**.**jugador**.actualizar();  **this**.**contrario**.actualizar(); |

Debemos dibujarlo.

|  |
| --- |
| dibujar (){   **this**.**fondo**.dibujar();  **for** (**var** i=0; i < **this**.**fondos**.length; i++){  **this**.**fondos**[i].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**.**contrario**.dibujar(**this**.**scrollX**);  **this**.**jugador**.dibujar(**this**.**scrollX**); |

Y también hay que tener en cuenta al **contrario** en el método **procesarControles().** El código que incluimos es exactamente igual al del Jugador, pero teniendo en cuenta la variable **controles2** y **contrario**.

|  |
| --- |
| procesarControles( ){  *// disparar* **if** ( ***controles2***.**disparo** ){  **var** nuevoDisparo = **this**.**contrario**.disparar();  **if** ( nuevoDisparo != **null** ) {  nuevoDisparo.configurarEspacio(**this**.**space**);  **if** ( **this**.**contrario**.**orientacion** == ***orientaciones***.**izquierda** ){  nuevoDisparo.body.applyImpulse(**cp**.v(-0.35, -0.1), **cp**.v(0, 0));  nuevoDisparo.body.p.**x** = nuevoDisparo.body.p.**x** - 20;  } **else** {  nuevoDisparo.body.applyImpulse(**cp**.v(0.35, -0.1), **cp**.v(0, 0));  nuevoDisparo.body.p.**x** = nuevoDisparo.body.p.**x** + 20;  }  **this**.**disparosJugador**.push(nuevoDisparo);  }  }   *// Eje X* **if** ( ***controles2***.**moverX** > 0 ){  **this**.**contrario**.moverX(1);  }**else if** ( ***controles2***.**moverX** < 0){  **this**.**contrario**.moverX(-1);  } **else** {  **this**.**contrario**.moverX(0);  }  *// Eje Y* **if** ( ***controles2***.**moverY** > 0 ){  **this**.**contrario**.saltar();  }     *// disparar* **if** ( ***controles***.**disparo** ){  **var** nuevoDisparo = **this**.**jugador**.disparar(); |

Ejecutamos el juego y comprobamos que ambos jugadores se visualizan y se mueven correctamente.



# Habilitar controles en mando (si queremos usar dos mandos)

Abrimos el fichero **EventosGamePad.js.** Creamos un segundo array para los botones que están actualmente pulsados en el mando 2.

|  |
| --- |
| ***console***.log(**"Navegador admiteMandos: "**+*admiteMandos*());  *// Los que hay pulsados actualmente* **var *botonesPulsados*** = []; **var *botonesPulsadosMando2*** = []; |

En el **addEventListener** creamos otro timer (similar a un hilo que ejecuta un while infinito) que obtenga las pulsaciones, en este caso del mando2.

|  |
| --- |
| ***window***.addEventListener(**"gamepadconnected"**, **function**(e) {  ***console***.log(**"GamePad Conectado tipo: "**+e.gamepad.id);  *// Se podrían conectar varios, multijugador* ***console***.log(**"Número: "**+e.gamepad.index);   *// Leer botones del mando* **if** ( ***navigator***.getGamepads()[0] != **null** ){  *setInterval*(*actualizarOrdenes*,1000/30);  }  **if** ( ***navigator***.getGamepads()[1] != **null** ){  *setInterval*(actualizarOrdenesMando2,1000/30);  } }); |

Agregamos la función **actualizarOrdenes2()** . Sera casi idéntica a la función que procesa las pulsaciones del mando1. Las diferencias se encuentran en que lee el mando 2 y utiliza la lista de botones botonesPulsadosMando2 y escribe las acciones sobre la variable controles2.

|  |
| --- |
| **function** *actualizarOrdenesMando2*(){  *// Obtener gamePad en cada iteración* **var** gP2 = ***navigator***.getGamepads()[1];   **for**(**var** i=0; i < gP2.buttons.length; i++){  **if** ( gP2.buttons[i].pressed ) {  ***entrada*** = ***entradas***.**gamepad**;  }  }  **if** ( **entrada** != ***entradas***.**gamepad**){  **return**;  }  *// Float con 2 decimales* ***controles2***.**moverX** = *parseFloat*(gP2.axes[0]).toFixed( 2 );   **if** ( gP2.buttons[2].pressed){ *// 0 es el botón X* **if** ( ***botonesPulsadosMando2***[2] == **false** ) {  ***botonesPulsadosMando2***[2] = **true**;  ***controles2***.**moverY** = 1;  }  } **else** {  ***botonesPulsadosMando2***[2] = **false**;  ***controles2***.**moverY** = 0;  }   **if** ( gP2.buttons[3].pressed){ *// 2 es el botón A* **if** ( ***botonesPulsadosMando2***[3] == **false** ) {  ***botonesPulsadosMando2***[3] = **true  *controles2***.**disparo** = **true**;  ***controles2***.**continuar** = **true**;  }  } **else** {  ***botonesPulsadosMando2***[3] = **false  *controles2***.**disparo** = **false**;  ***controles2***.**continuar** = **false**;  } } |

# Perder

Abrimos el **GameLayer.js** en cada iteración del **actualizar()** vamos a comprobar sí el **jugador** o el **competidor** tienen una posición **y** menor de 520 (realmente se sale del mapa en la 320, la aumentamos un poco para que se caiga durante unos segundos).

Podemos utilizar tanto **this.jugador.y** como **this.jugador.body.p.y** ya que ambas coordenadas se igualan en cada ejecución del actualizar del jugador.

|  |
| --- |
| actualizar (){  **this**.**space**.step(1000/30);   **if** ( **this**.**jugador**.**body**.p.**y** > 520 || **this**.**contrario**.**body**.p.**y** > 520){  **this**.iniciar();  }    *// actualizar* |

# Colisiones

Chipmunk permite registrar colisiones entre formas (shapes) del espacio mediante escuchadores. El primer paso es incluir identificadores a los elementos, para que el espacio sepa a qué elemento corresponde cada forma.

Vamos a crear estos identificadores en el fichero **Globales.js**

|  |
| --- |
| **var *pulsaciones*** = []; *// actuales registradas* **var *cJugador*** = 1; **var *cDisparo*** = 2; |

Debemos asignar el identificador en el momento de crear las formas (shapes), comenzamos por **Jugador.js** en su método **configurarEspacio**

|  |
| --- |
| configurarEspacio(space){   *//1 Crear cuerpo* **this**.**body** = **new cp**.Body(3, ***Infinity***);  space.addBody(**this**.**body**);   *//posicionar cuerpo (El cuerpo tiene la posición)* **this**.**body**.**p**.**x** = **this**.**x**;  **this**.**body**.**p**.**y** = **this**.**y**;   *//2 Crear forma (asociada al body)* **this**.**shape** = **new cp**.BoxShape(**this**.**body**,  **this**.**ancho**, **this**.**alto**);  **this**.**shape**.setCollisionType(***cJugador***);  **this**.**shape**.setFriction(0.8); |

Repetimos lo mismo en el **DisparoJugador.js** , añadiendo el tipo **cDisparo**.

|  |
| --- |
| configurarEspacio(space) {  *//1 Crear cuerpo* **this**.**body** = **new cp**.Body(1, ***Infinity***);  space.addBody(**this**.**body**);   *//posicionar cuerpo (El cuerpo tiene la posición)* **this**.**body**.**p**.**x** = **this**.**x**;  **this**.**body**.**p**.**y** = **this**.**y**;   *//2 Crear forma (asociada al body)* **this**.**shape** = **new cp**.BoxShape(**this**.**body**,  **this**.**ancho**, **this**.**alto**);  **this**.**shape**.setCollisionType(***cDisparo***);  **this**.**shape**.setElasticity(0.6); |

Ahora ya podemos registrar colisiones entre formas de tipo **cJugador** y **cDisparo**, para ello abrimos el **GameLayer.js** y nos colocamos en el metodo **iniciar(),** después de instanciar el **espacio** podemos agregar escuchadores.

Primero se definen los tipos de elementos para los que se va a registrar el choque, y luego la función que se va a ejecutar, en este caso **collisionJugadorDisparo** (aun sin implementar)

|  |
| --- |
| iniciar() {   **this**.**space** = **new cp**.Space();  **this**.**space**.**gravity** = **cp**.v(0, 0.0006);   **this**.**space**.addCollisionHandler(***cJugador***,***cDisparo***,  **null**, **null**, **this**.collisionJugadorDisparo.bind(**this**), **null**); |

|  |
| --- |
| La función **addCollisionHandler** puede recibir varios posibles parámetros   1. **ID forma 1** 2. **ID forma 2** 3. **Inicio:**  función de callback nos avisa justo antes de chocar. 4. **Antes de resolver la colisión:** función de callback nos avisa antes de resolver la colisión física (repartir fuerzas en el golpe) 5. **Después de resolver la colisión:** función de callback nos avisa después de resolver la colisión física 6. **En la separación:** función de callback que nos avisa cuando las formas se separen |

Está función va a eliminar los disparos, pero eso no se puede hacer dentro de la propia función, ya que se ejecuta dentro de la función **space.step()** y podría ser que un elemento se eliminase mientras está siendo recorrido.

Vamos a utilizar una lista auxiliar **disparosEliminar** que nos indique los disparos que vamos a eliminar.

|  |
| --- |
| collisionJugadorDisparo (arbiter, space){  **var** shapes = arbiter.getShapes();    *// shapes[0] es la forma del jugador  // shapes[1] es la forma del disparo* **if** (**this**.disparosEliminar.indexOf(shapes[1].body.**modelo**) == -1) {  **this**.disparosEliminar.push(shapes[1].body.**modelo**);  } } |

Creamos la lista **disparosEliminar** dentro del **iniciar().**

|  |
| --- |
| iniciar() {  **this**.**disparosEliminar** = [];    **this**.**space** = **new cp**.Space();  **this**.**space**.**gravity** = **cp**.v(0, 0.0006); |

En cada iteración del **actualizar()**, vamos a recorrer la lista **disparosEliminar** y sacar los disparos del juego y del espacio (el espacio incluye el cuerpo y la forma).

Una vez hemos borrado todos los **disparosEliminar** podemos vaciar el array.

|  |
| --- |
| actualizar (){  **this**.**space**.step(1000/30);   **for**(**var** i = 0; i < **this**.**disparosEliminar**.length; i++) {    **var** j = **this**.**disparosJugador**.indexOf(**this**.**disparosEliminar**[i]);  *// quita la forma* **this**.**space**.removeShape(**this**.**disparosJugador**[j].body.**shapeList**[0]);  *// quita el cuerpo \*opcional, funciona igual* **this**.**space**.removeBody(**this**.**disparosJugador**[j].body);  *// quita el disparo del juego* **this**.**disparosJugador**.splice(j, 1);   }  **this**.**disparosEliminar** = []; |

Ejecutamos el juego y comprobamos que los disparos desaparecen al chocar con los jugadores.



# Jugador Bloqueado

Vamos a incluir un sistema que deje bloqueado al jugador durante 34 segundos cada vez que colisione con un disparo. La función **collisionJugadorDisparo**, accederá al jugador y le colocará la variable **bloqueado** a 90.

|  |
| --- |
| collisionJugadorDisparo (arbiter, space){  **var** shapes = arbiter.getShapes();   *// shapes[0] es la forma del jugador  // shapes[1] es la forma del disparo* shapes[0].body.**modelo**.**bloqueado** = 60;    **if** (**this**.**disparosEliminar**.indexOf(shapes[1].body.**modelo**) == -1) {  **this**.**disparosEliminar**.push(shapes[1].body.**modelo**);  } } |

Accedemos al **Jugador.js,** en su constructor, iniciamos la variable **bloqueado** a **0** (ausencia de bloqueo).

|  |
| --- |
| constructor(x, y) {  **super**(***imagenes***.**jugador** , x, y)  **this**.**quieto**=0;  **this**.**bloqueado** = 0; |

En cada iteración del **actualizar()** se reducirá el valor de **this.bloqueado**, si este es mayor de 0.

|  |
| --- |
| actualizar(){  **if** ( **this**.**bloqueado** > 0) {  **this**.**bloqueado** = **this**.**bloqueado** - 1;  }    **this**.**x** = **this**.**body**.**p**.**x**; |

Vamos a hacer que **moverX(direccion)** no funcione sí el jugador tiene tiempo de bloqueo.

|  |
| --- |
| moverX (direccion){  **if** (**this**.**bloqueado** > 0 )  **return   if** (direccion > 0 && **this**.**body**.**vx** < 0.15) {  **this**.**body**.applyImpulse(**cp**.v(0.15, 0.08), **cp**.v(0, 0));  } |

Aplicamos la misma restricción en **disparar().**

|  |
| --- |
| disparar(){  **if** (**this**.**bloqueado** > 0 )  **return    if** ( **this**.**tiempoDisparo** == 0) {  *// reiniciar Cadencia* **this**.**estado** = ***estados***.**disparando**; |

También en **saltar().**

|  |
| --- |
| saltar(){  **if** (**this**.**bloqueado** > 0 )  **return    if** ( !**this**.**enElAire** ) {  **this**.**body**.applyImpulse(**cp**.v(0, -1.2), **cp**.v(0, 0)); |

Por ultimo modificamos el **dibujar(),** para hacer que el jugador solo se dibuje cuando no tiene tiempo de bloqueo, o sí lo tiene cuando este sea múltiplo de 3 (así parpadeará)

|  |
| --- |
| dibujar (scrollX){  **if** (**this**.**bloqueado** <= 0 || **this**.**bloqueado** % 3) {  scrollX = scrollX || 0;  **this**.**animacion**.dibujar(**this**.**x** - scrollX, **this**.**y**);  } } |

Ejecutamos el juego y comprobamos que el bloqueo funciona correctamente.

