

Práctica: Super Pang

Juanan Pereira

15 de marzo de 2020



Índice

1. Introducción y reglas básicas	1
1.1. Interacción del usuario	1
1.2. Burbujas	1
1.3. Panel de puntuación	1
1.4. Niveles	1
2. Cargando a Buster. Clase SpriteSheet	2
3. Timers y Sprites	3
4. Control de teclado	3
5. Moviendo a Buster	4
6. Animando a Buster	5
7. Niveles y JSON	5
8. Creando burbujas	6
9. Hooks: disparando contra las bolas	6
10. Backgrounds	9
11. Detalles	10
12. Volteando imágenes	12
13. Sprites en las bolas	13
14. Hackeando la gravedad	15
15. Gestión de niveles	16
16. Bonus	17
17. Efectos de sonido	18
18. Game Over	19
19. Final	19
19.1. Niveles	19
19.2. Tiempo	19
19.3. Obstáculos	19
19.4. Bonus	20
19.5. Puntuación	20
19.6. Hall of Fame	20
19.7. Soporte multibrowser	20
19.8. Soporte del GamePad API	20

1. Introducción y reglas básicas

Super Buster Bros., publicado como Super Pang fuera de América del Norte, es un videojuego cooperativo de dos jugadores, desarrollado por Mitchell y lanzado en los Estados Unidos en 1990 por Capcom. Es el segundo juego de la serie Pang y fue portado a Super Nintendo en 1992.

El objetivo del juego es usar una pistola de arpones para hacer estallar las burbujas o bolas que rebotan en la pantalla. Cuando un jugador hace estallar una burbuja, se divide en dos burbujas más pequeñas. Las burbujas suficientemente pequeñas simplemente se vaporizan cuando revientan. Aunque el juego de arcade (máquina recreativa) y la versión de PlayStation permiten que dos jugadores jueguen simultáneamente, la versión Super Nintendo (y la que nosotros programaremos) solo tiene un jugador. Hay *power-ups* repartidos por los niveles, haciendo estallar ciertas burbujas, disparando a cajas o disparando a ciertos puntos sin marcar en el nivel.

En el juego original, hay dos modos, Panic y Tour. Implementaremos el modo Tour, donde debemos ayudar a Buster a avanzar de nivel en nivel por países de todo el mundo. En cada uno, deberemos eliminar todas las burbujas antes de que el tiempo se acabe, sin que ninguna bola toque a nuestro jugador, en cuyo caso perderemos una vida.

1.1. Interacción del usuario

El usuario puede usar las flechas del teclado (, , , )¹ para mover a Buster.

1.2. Burbujas

En el juego original, cada país representado debe ser atravesado en 3 pantallas que representan al país por la mañana, por la tarde y por la noche. Durante la mañana las burbujas eran rojas, por la tarde azules y por la noche, verdes.

1.3. Panel de puntuación

El panel de puntuación reflejará, aparte de los puntos acumulados, la puntuación máxima obtenida hasta el momento por cualquier jugador que hubiera jugado previamente al Pang. En una primera versión, mostraremos también el número de vidas restantes. En la versión final, las vidas se mostrarán representadas gráficamente por medio de pequeños Buster.

La gestión de la puntuación se llevará a cabo en la segunda parte de la práctica.

1.4. Niveles

Antes de comenzar a jugar hay que visualizar el estado inicial de la pantalla (el mapa de cada nivel). Para ello, el primer paso será construir dicho mapa, que puede ser diferente para cada uno de los niveles (como en el juego original).

¹ y,  sólo si implementas la opción de añadir escaleras al juego, tal y como ocurre en el original



En las siguientes secciones se irá preguntando al respecto del desarrollo de la práctica de SuperPang siguiendo los vídeos indicados^a. En cada sección, debes responder a las preguntas planteadas y subir una versión de tu código a un repositorio privado en GitHub. Puedes hacer tantos commits y push como quieras, pero debe haber commits que tengan un tag identificativo ^b. Los nombres de los tags se indicarán a lo largo de la práctica. Un script recogerá automáticamente los commits que estén etiquetados con el tag y comprobará que el código es correcto.

^aGran parte de los vídeos están basados en la serie desarrollada por Meth Meth Method para la construcción del juego de SuperMario Bros en JS <https://www.youtube.com/watch?v=g-FpDQ8Eqw8>

^bSi nunca los has usado, puedes aprender a crear tags en un repositorio Git siguiendo este tutorial: <https://git-scm.com/book/en/v2/Git-Basics-Tagging> o echarle un vistazo al vídeo que se publicará al respecto en eGela

2. Cargando a Buster. Clase SpriteSheet

Estudia los vídeos 1 y 2 y responde a las siguientes preguntas.

1. ¿Qué significan los parámetros del método `context.drawImage(p1,p2,p3,p4,p5,p6,p7,p8,p9)`?
2. ¿Qué tipo de datos acepta el método `context.drawImage` en su primer parámetro?
3. ¿Qué hace el método `define(x,y,z)` de `SpriteSheet`?
4. ¿Qué hace el método `draw(x1,x2,x3,x4)` de la clase `SpriteSheet`?
5. **Descarga** la hoja de sprites de SuperPang (para el jugador).
6. Convierte el fondo rosa en transparente (puedes usar esta **herramienta online**(Ver Fig. 1)
7. Carga el sprite del jugador Buster (32x32) de la figura, para conseguir mostrar a Buster en pantalla.
8. **Crea una etiqueta Buster** en este punto dentro de tu historial de commits Git.
9. Supongamos que quiero crear la siguiente estructura de datos:

```
{ "a" => 1, "b" => 2 }
```

¿Cuál sería el código JavaScript que la genera? ¿Cuál sería el código JavaScript que permite imprimir los valores así?²

```
Clave: "a" , Valor: 1  
Clave: "b" , Valor: 2
```

²Ayuda: <https://hackernoon.com/what-you-should-know-about-es6-maps-dc66af6b9a1e>



Si paras el servidor *server* y lo quieres volver a lanzar, basta con que uses el comando `npm start`. Si quieres saber más sobre Server.js, échale un vistazo a su [web oficial](#).



Figura 1: Debes convertir el fondo de la hoja de Sprites a un color transparente



Figura 2: Colocando a Buster en pantalla

3. Timers y Sprites

Revisa el [vídeo 3](#) y [el ejercicio asociado](#) para responder a las siguientes preguntas.

Crea y **exporta** las clases `Vec2D` y `Object2D`. Guárdalas en un fichero de nombre `math.js`. Descarga también `Player.js` y `Settings.js`. Completa el código de los métodos de la clase `Vec2D` y de la clase `Player` que se han esbozado con pseudocódigo.

4. Control de teclado

Revisa con atención el [vídeo 4](#) sobre Gestión de teclado.

Si pulsamos cualquier tecla se dispara un evento `keydown` que podremos capturar. Por ejemplo, si pulsamos la tecla "j":

```
KeyboardEvent {isTrusted: true, key: "j", code: "KeyJ",
location: 0, ctrlKey: false, }
```

Vemos que el nombre de código que le asigna es "KeyJ". y El keycode es el 74.

1. ¿Cuál es el nombre de código de tecla para el número 0 del teclado? ¿Y para la tecla Alt izquierda? (si estás en macOS, para la tecla Command izquierda).
2. ¿Qué desventajas se nombran para no usar un gestor de eventos JS para gestionar la pulsación de teclado?
3. ¿Por qué necesitamos llamar a este método `event.preventDefault()` en la función `handleEvent`?
4. ¿Qué hace exactamente esta línea de código? `const {code} = event;`
5. Queremos reescribir esta línea de código sin el operador condicional ternario. ¿Cómo debemos hacerlo?³ `const keyState = event.type === 'keydown' ? PRESSED : RELEASED;`
6. ¿Qué tipo de variable recibe el segundo parámetro de esta función?
`addMapping(keyCode, callback)`
7. En 10:20 hay un error de continuidad. Fíjate que se introduce un `console.log(this.keyStates)` en la línea 35 de `Keyboard.js` sin decir nada al respecto. ¿Para qué? (¿qué se mostraría en pantalla, al probar la clase al final del vídeo, si no hubiera ese `console.log`?)
8. al pulsar la barra espaciadora se generan dos eventos (`keydown`, `keyup`). Si pulsar espacio es equivalente a disparar, estaremos disparando dos veces. ¿Qué cambiarías en el código para que sólo se dispare al pulsar la tecla (no al soltarla)?

5. Moviendo a Buster

Para poder mover a Buster con las flechas del teclado, debemos añadir código para gestionar los eventos de teclado, es decir, conocer cuándo el usuario ha pulsado una tecla. Hay que preparar algunas tareas antes de ponernos a ello:

1. modifica el tamaño del canvas para que el tamaño de la pantalla sea (400x300).
2. añade código a `main` para inicializar los valores de `SCREEN_HEIGHT` y `SCREEN_WIDTH` al tamaño del canvas.
3. Pinta una línea negra alrededor del canvas para conocer en todo momento, de manera visual, sus dimensiones.
4. Coloca a Buster en la parte inferior central del canvas (debe empezar el juego con Busterparado).
5. Extra el código de gestión de teclado de `main.js` a una función `setupKeyboard` (expórtala desde el fichero `input.js`), para que quede así:

```
const input = setupKeyboard(buster);
input.listenTo(window);
```

6. Añade el código necesario para capturar el **keydown** de las teclas \leftarrow , \uparrow , \rightarrow y \downarrow (debes mostrar en consola un mensaje con el nombre de la tecla pulsada, ver Fig. 3).

³https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

7. Cambia la dirección en la que se mueve Buster en función de la tecla pulsada. Recuerda que Player tiene un atributo direction...
8. Comprueba que puedes mover a Buster de izquierda a derecha en pantalla
9. **Crea una nueva etiqueta Movement** en el código Git de tu juego.

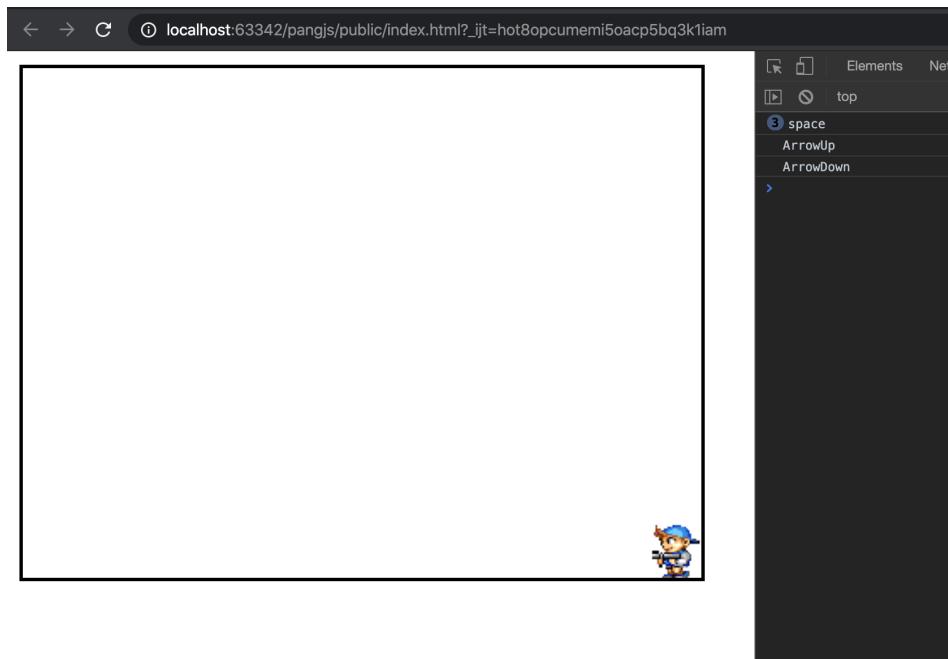


Figura 3: Gestionando el teclado

6. Animando a Buster

Revisa el [vídeo 5](#) y crea un tag tal y como se indica:

1. **Crea una etiqueta BusterAnimation** en este punto dentro de tu historial de commits Git.
2. Vemos que Buster camina siempre hacia la izquierda. Modificaremos la clase SpriteSheet para crear dos sprites de cada posición de Buster: la posición original y la posición espejo, pero eso será en la segunda parte de la práctica.

7. Niveles y JSON

Revisa el [vídeo 6](#) y responde a los siguientes ejercicios y preguntas.

1. Para leer el contenido de un fichero JSON se usa la siguiente función:

```
function loadLevel(name) {
    return fetch(`/levels/${name}.json`)
        .then(r => r.json());}
```

En el parámetro de `fetch` se usan [Template literals](#). Supongamos que usamos una versión de ECMAScript anterior a ES2015 (y por tanto, no podemos usar Template Literals). ¿Cómo reescribirías el código para que su funcionamiento fuera el mismo?

2. Los métodos `loadImage` y `loadLevel`, se ejecutan en serie o en paralelo?

3. ¿Para qué se usa esta construcción JS?

```
Promise.all([
  ...
])
.then(([...]) => {
  ...
});
```

8. Creando burbujas

Añade el método `draw` a la [clase Ball](#)⁴ para que se pinte a sí misma en el contexto que se le pasa como parámetro. Fíjate en dos detalles:

1. Dispones del método `update` ya programado, que modifica la posición y fuerza en la que se mueve la burbuja en función del tiempo transcurrido desde la última vez que se llamó al método.
2. El método `update` hace uso de la clase `Settings`, donde se definen ciertas variables, entre ellas, `SCREEN_WIDTH` y `SCREEN_HEIGHT`. Estos dos valores deben calcularse desde `main.js`, en función del tamaño del canvas.

Finalmente,

1. En `main.js`, tras cargar a Buster, carga las bolas del nivel usando este método:

```
const balls = loadBalls(levelSpec.balls);
```

2. implementa el método `loadBalls` en `loaders.js`. Este método debe instanciar uno objeto `Ball` por cada bola del JSON que se le pase como parámetro y devolver un array con todas la bolas instanciadas.

3. En el método `update` de `main.js`, debes pintar y actualizar la posición de cada bola en pantalla.

4. **Crea una etiqueta Burbuja** en este punto dentro de tu historial de commits Git.

Si todo va bien, debería de ver una pantalla como la siguiente (Fig. 4), con las burbujas rebotando a lo largo de la pantalla.

9. Hooks: disparando contra las bolas

Ya tenemos todo preparado para empezar a explotar burbujas. Sólo nos falta añadir soporte para disparar ganchos (hooks) (Fig. 5). Para ello, importa [Hook.js](#) en tu proyecto y completa el código que encontrarás comentado. A continuación, en `main.js`:

- Carga la imagen `img/hookRope.png`

⁴Ball.js hace uso de `Object2D` y de `Settings`



Figura 4: La burbuja debe rebotar por las paredes del canvas.



(a) Lo que hemos hecho (¡y el suelo aún está sin añadir!) (b) Lo que nos queda por hacer

Figura 5: Comenzamos a añadir soporte para disparar a las bolas.

- modifica el código de main para introducir un array de hooks y un cargador de hooks (loadHookManager es una función de orden superior que toma como parámetros la imagen del hook y el array de hooks en pantalla). La función devuelta se la asignaremos a Buster a través del método setHookManager (tendrás que programar tanto loadHookManager como setHookManager). hookManager es una función que toma como parámetros la posición x,y donde hay que lanzar el hook y crea un nuevo hook (initialmente de tipo rope) en esa posición, añadiéndolo al array hooks). No debes añadir ningún otro método a main.js.

```
const hooks = [];  
const hookManager = loadHookManager(hookImage, hooks);  
const buster = loadBuster(playerImage, levelSpec.player);  
buster.setHookManager(hookManager);
```

- en el método update() añade dos líneas para el draw y el update de los hooks

En input.js:

- al pulsar espacio, disparar un nuevo hook (llamar al método shoot de Player)

- el método shoot llama a hookManager pasándole como parámetros la posición central de Buster (la posición desde donde lanzar el hook)

Si dejamos el código tal cual, podremos lanzar ganchos, pero estos quedarán pegados en la parte superior de la pantalla, podremos lanzar tantos como queramos y no romperán bolas (porque aún no hemos programado el control de colisiones) (Fig. 6).

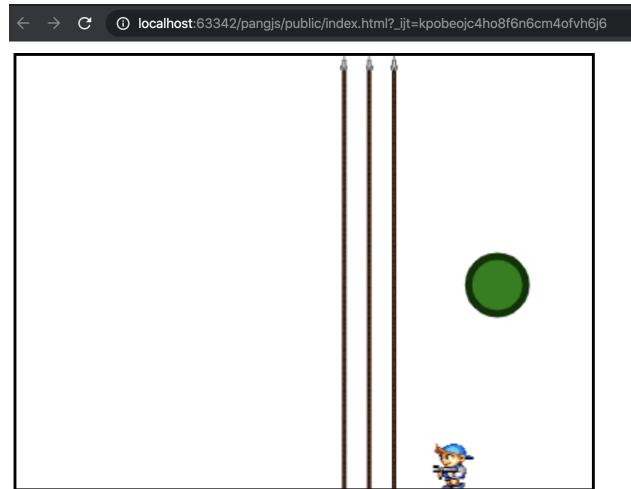


Figura 6: Lanzamos ganchos y se ve cómo van subiendo, pero estos quedan clavados al techo y encima no rompen burbujas... hay que mejorar el código.

Importa el fichero **collisions.js** que incluye la función `ball_to_box` para detectar colisiones entre la bola y otra entidad.

- Implementa una clase CollisionManager dentro de collisions.js que tome como parámetros del constructor el array de hooks y el array de balls. Debe ofrecer un método `checkCollisions` que compruebe si hay choque entre alguna bola y algún hook. En tal caso, que imprima en consola el string "pang!"
- En main.js, añade una llamada a la función `check_collisions` de collisionManager.
- **Crea una etiqueta HooksCollision** en este punto dentro de tu historial de commits Git.
- Añade esta constante a Settings.js (indica cuál es el radio mínimo de una bola. Si rompemos una bola de este radio, debe desaparecer)

```
static MIN_BALL_RADIUS = 5;
```

- añade el método `split_ball` a CollisionManager. Toma como parámetro una bola y si su radio es mayor que `Settings.MIN_BALL_RADIUS` la divide en dos subolas de mitad de radio. Una va hacia la izquierda y otra hacia la derecha.
- en el método `check_collisions`, si se detecta una colisión, ahora dividirá la bola en dos y eliminará el hook y la bola pertinente del juego⁵

⁵Para borrar cómodamente una bola o un hook del juego, sería MUY recomendable sustituir los arrays por Sets, que incluyen un método `delete...`

- Para eliminar los ganchos una vez que lleguen a tocar el techo de la pantalla, añade una condición `if (hook.to_kill)` al método `check_collisions` de `CollisionManager` (recuerda que este atributo `to_kill` se calcula en `Hook.js` cuando el gancho supera el techo). Si se cumple la condición, elimina el gancho.
- en la función `hookManager`, añade una comprobación para no permitir crear nuevos hooks si en pantalla ya hay `Settings.MAX_HOOKS` (crea esta variable y dale un valor razonable)
- **Crea una etiqueta HookManager** en este punto dentro de tu historial de commits Git.

10. Backgrounds

El fondo blanco es muy soso. Vamos a darle un poco de vida. Lo primero será **importar el fichero** con todos los fondos de pantalla disponibles en el juego original.

En la imagen hay 40 pantallas de 240x192 pixels cada una (Fig. 7).



Figura 7: Cada pantalla, salvo la primera, representa una ciudad por la mañana, tarde y noche.

La siguiente función de orden superior `loadBackground`, extrae la pantalla 0 del fichero de backgrounds y devuelve una función que permite pintarla en el contexto que le llegue como parámetro (no tiene en cuenta el nivel en el que estamos, siempre extrae y pinta el nivel 0).

```
export function loadBackground(backgrounds) {
  const buffer = document.createElement('canvas');
  buffer.width = 256;
  buffer.height = 192;

  // recortar super-sprite y dejarlo preparado en un buffer
  const context = buffer.getContext("2d");
  context.drawImage(backgrounds, 0, 0,
    buffer.width, buffer.height,
    0, 0, buffer.width, buffer.height,);

  return function (ctx) {
    ctx.drawImage(buffer,
      0, 0,
      buffer.width, buffer.height,
      0, 0,
      Settings.SCREEN_WIDTH, Settings.SCREEN_HEIGHT);
  }
}
```

Listing 1: La función `loadBackground` devuelve otra función

1. En el `Promise.all` de `main.js` introduce otra promesa para cargar la imagen de `backgrounds`.

2. Cuando la promesa se resuelva, llama al método loadBackground indicado

```
const drawBackground = loadBackground(backgrounds);
```

y finalmente, usa drawBackground para pintar el background en el contexto del juego (elimina el clearRect que había en su lugar)

3. Debería verse un resultado como el de la Fig. 8.

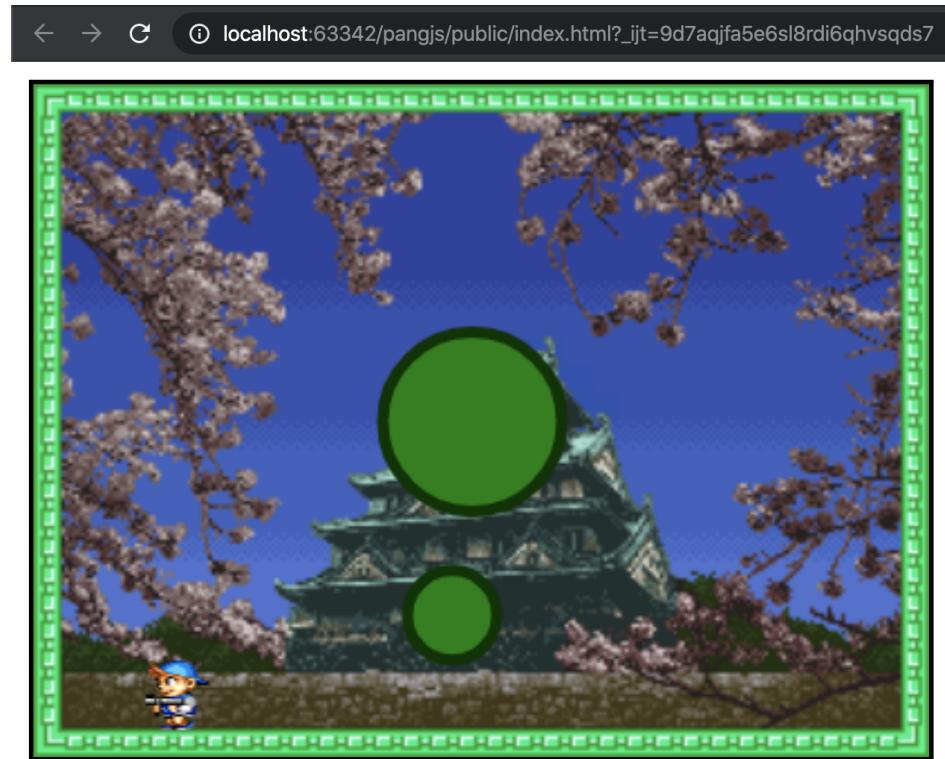


Figura 8: Ya casi hemos terminado con la primera parte de la práctica.

11. Detalles

Algunos detalles más antes de terminar.

1. Modifica Player.js para que Buster se pinte justo encima del suelo. Hay 12 pixels de margen (Settings.MARGIN lleva ese valor).
2. Buster debe ir más rápido (Settings.PLAYER_SPEED = 200 parece razonable)
3. Hay que refactorizar la clase Ball para que las bolas reboten mucho más rápido. Te recomiendo que uses [esta implementación alternativa](#) de Ball.js (tendrás que refactorizar collision.js para usar los nombres de variables que modifiques en Ball.js, así como la función loadBalls de loaders.js)
4. Si sigues la implementación del punto anterior, tendrás que reducir la fuerza con la que salen las bolas inicialmente.

```

    {
    "player": {
        "pos" : [184, 253]
    },
    "balls": [
        {
            "radius": 20,
            "pos": [100,100],
            "force": [1,0.5]
        },
        {
            "radius": 40,
            "pos": [50,50],
            "force": [1,0.5]
        }
    ]
}

```

5. El método routeFrame debe devolver el sprite de Buster en el que está mirando hacia arriba (vamos a llamarle 'idle'). Así, cuando Buster no esté andando quedará mirando hacia arriba. Carga el buffer de idle en loaders.js.
6. **Crea una etiqueta Parte1** en el código git de tu práctica.

Puedes ver en [este tweet](#) cómo debe quedar el resultado final (aprox.). Siguiendo el hilo de tweets, podrás ver la evolución del juego.



¡ENHORABUENA!

Has terminado de programar un prototipo reconocible del juego SuperPang :-) Tómate un descanso, te lo has ganado. Con lo que has trabajado, has conseguido aprobar la práctica pero... si buscas aprender a programar juegos profesionales (y obtener una buena nota en la práctica :) te animo a que sigas trabajando un poco más. Aún hay muchas funcionalidades que debemos añadir (sprites para las bolas, música y efectos de sonido, puntuación, gestión de niveles, colisión Bola-Buster) y otras que debemos corregir (refactorizar el código para evitar duplicidades, hacerlo más comprensible y más fácil de mantener en el futuro). También plantearemos retos, totalmente opcionales (bonus y bichos en pantalla, poder disparar, congelar las bolas, jugar con un mando...)

La siguiente parte de la práctica no estará tan guiada. Esto tiene el inconveniente de que trabajarás sin una red de protección, y la ventaja de que serás libre de programar el código que consideres más adecuado para cada ejercicio. ¿Serás capaz de superar este reto? Lo sabremos dentro de unos pocos días...

12. Volteando imágenes

La animación de Buster siempre camina hacia la izquierda, aunque nos estemos moviendo al lado contrario. Si nos fijamos en los sprites que forman player.png, no tenemos la imagen de Buster caminando hacia la derecha. Pero si pudiéramos implementar una función espejo, ya tendríamos solucionado nuestro problema :) Eso es precisamente lo que vamos a hacer. Para cada sprite que nos interese, le aplicaremos una función espejo y así tendremos el sprite en las dos posiciones. Ver Fig. 9.

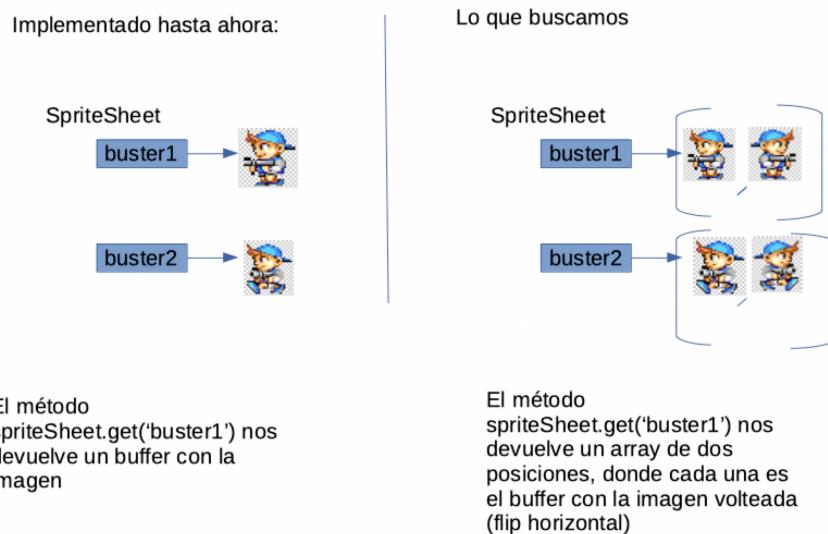


Figura 9: Buscamos una función espejo que nos permita tener los sprites de Buster volteados.

Para voltear una imagen vamos a usar un truco del canvas.

```
canvas = document.createElement('canvas');
canvasContext = canvas.getContext('2d');

canvasContext.translate(width, 0);
canvasContext.scale(-1, 1);
this.canvasContext.drawImage(image, 0, 0);
```

Las líneas 4 y 5 permiten hacer un flip horizontal de una imagen haciendo uso de los métodos `translate` y `scale`.

Sabiendo eso, podemos modificar el método `define` para que en lugar de crear un sprite para cada elemento, cree un array de dos sprites (original y flip). Haremos uso de la función `map`. Esta función es muy similar a un `forEach`, salvo que va acumulando los resultados en un array (que devuelve al finalizar).

```

define(name, x, y) {
    const buffers = [false, true].map( flip => {
        const buffer = document.createElement('canvas');
        buffer.width = this.width;
        buffer.height = this.height;
        const context = buffer.getContext('2d');
        if (flip) {
            context.scale(-1,1);
            context.translate(-buffer.width, 0);
        }
        context
            .drawImage(this.image,
                x * this.width, y * this.height,
                this.width, this.height,
                0, 0,
                this.width, this.height);
        return buffer;
    });
    this.sprites.set(name, buffers);
}

```

El método get de SpriteSheet también tendremos que modificarlo:

```

get(name) {
    return this.sprites.get(name);
}

```

Hasta ahora devolvía un buffer con la imagen. Ahora debe seguir haciendo lo mismo, pero de las dos que tenemos, ¿cuál devolvemos? Necesitamos un parámetro que nos diga si Buster se dirige hacia la izquierda o hacia la derecha y devolver la imagen original o la imagen espejo en función de ese valor.

```

get(name, heading) {
    return this.sprites.get(name) [heading>0?1:0];
}

```

1. Modifica el método draw() de Player.js para pasarle la dirección del jugador al método get anterior (recuerda que la dirección horizontal está guardada en el atributo direction.x.)
2. En el método get de SpriteSheet, indica que el valor por defecto del parámetro heading es 0

13. Sprites en las bolas

Dibujar una bola usando los métodos básicos del canvas (como *arc*) en cada vuelta del update, además de ser algo innecesario, no queda estéticamente bien. En su lugar, haremos uso de nuevos sprites (ver Fig. 11)⁶. Vamos allá.

1. En Ball.js, introduce dos nuevos atributos: color y sprite (llegan como parámetro al constructor). En el método draw, pintaremos este sprite (elimina el código que dibuja una bola usando arc).

⁶Disponibles aquí



Figura 10: Sprites en las bolas.



¡OJO! En el método draw() pintábamos con arc() las bolas en pantalla. En ese caso, (x,y) representa el centro de la bola. Si usas drawImage en su lugar (para pintar el sprite), el punto (x,y) ahora no representa lo mismo. Ténlo en cuenta. Si mantienes (x,y) sin cambiar, las colisiones no se detectarán bien...

2. Aunque disponemos del método constructor de Ball, este no sabe qué sprites usar⁷. El constructor de la bola recibirá el sprite, sin tener que preocuparse de seleccionarlo al pintar. Por otra parte, también creamos nuevas bolas (más pequeñas) en collisions.js (cuando un hook destruye una bola) y ahí también necesitamos pasarle el sprite. Para solucionar todo esto, vamos a crear una función que cree bolas (ballFactory) en loaders.js y determine qué sprite usar.

```

1  export function createBallFactory(ballsImage) {
2
3  const spriteSheet = new SpriteSheet(ballsImage, 48, 40);
4  spriteSheet.define('red20', 0, 0);
5  spriteSheet.define('blue20', 1, 0);
6  spriteSheet.define('green20', 2, 0);
7
8  return function (radius, pos, force, color) {
9
10     let sprite = `${color}${radius}`;
11
12     return new Ball(radius, pos, force, spriteSheet.get(sprite),
13         color);
14 }
```

Para crear la factoría de bolas necesitamos decirle dónde están los sprites de las mismas (parámetro de la línea 1). Sabiendo que nos llegará una imagen con varias bolas (roja, azul y verde) definimos los sprites adecuados (red20, blue20 y green20). La función que devolvemos (createBallFactory es una función de orden superior) recibe como parámetro el radio,

⁷Y cada sprite depende del color de la bola, que hasta ahora tampoco habíamos tenido en cuenta



Figura 11: Fichero balls.png incluye tres bolas, roja, azul, verde, de radio 20

posición y fuerza inicial, así como el color de la bola que queremos crear. Mediante un template string (línea 10), calculamos el nombre del sprite en nuestro spriteSheet y finalmente llamamos al constructor de Ball con todos los parámetros adecuados (fíjate que el sprite final lo obtenemos indexando spriteSheet con el nombre del sprite).

Las líneas 3 a 6 forman lo que se denomina cierre (closure) de la función que devolvemos. El cierre de la función, en este caso, nos sirve para encapsular los sprites. Fíjate que cuando la usemos, no le estamos pasando el spriteSheet y sin embargo, tendrá acceso al mismo.

3. Añade la definición de los sprites que faltan en createBallFactory.
4. En main.js, crearemos la factoría de bolas y se la pasaremos a aquellas funciones que necesiten crear nuevas bolas (loadBalls y CollisionManager)



El parámetro ballsImage de createBallFactory es una colección de imágenes que tendrás que cargar adecuadamente. Te recomiendo que empieces cargando sólo una imagen (las de radio 20) y luego (una vez que veas que funciona) refactorices el código para dar soporte a bolas de radio 10 y 5. He dejado también bolas de radio 40, pero en mi opinión son demasiado grandes. ¡A gusto del consumidor!

```
1 const ballFactory = createBallFactory(ballsImage);
2 const balls = loadBalls(levelSpec.balls, ballFactory);
3 ...
4
5 const collisionManager = new CollisionManager(hooks, balls, ballFactory);
```

5. en loaders.js (loadBalls) y en collisions.js (split_ball), reemplaza la llamada al constructor Ball por una llamada a ballFactory (necesitarás pasarle el color de la bola).
6. En levels/1.json añade un atributo “color” a cada bola que crees (posibles valores: red, green, blue). Si realizas todo lo anterior, debería ver una imagen en pantalla como la de la figura 10.

14. Hackeando la gravedad

La gravedad que aplicamos a las bolas en pantalla puede jugarnos una mala pasada si somos demasiado puristas. En [este tweet](#) podemos ver un vídeo en el que la gravedad hace que la bola vaya perdiendo altura en cada rebote. Esto provocará que, en cierto momento, la altura que alcanza la bola tras el rebote sea menor que la altura de Buster (incluso llegando a no rebotar). Lógicamente, si lo dejamos así, la dificultad del juego puede ser infernal. Necesitamos ajustar (hackear) la gravedad, para que la velocidad de la bola no pase de cierto umbral. Es decir, cancelaremos el efecto de la gravedad a partir de cierto punto. ¿Cómo calcular este punto? En general, ¿cómo visualizar valores

Lamentablemente esto no puede hacerse tan fácil en la vida real :)

que cambian rápidamente (la velocidad de la bola en este caso) sin tener que recurrir a `console.log`⁸? Lo que haremos será pintar en pantalla una línea de debug.

Modifica el método `draw` de `Ball.js` para que además de pintar la bola, pinte en la parte superior el valor de `vy` cuando rebota en la parte inferior (Fig. 12).

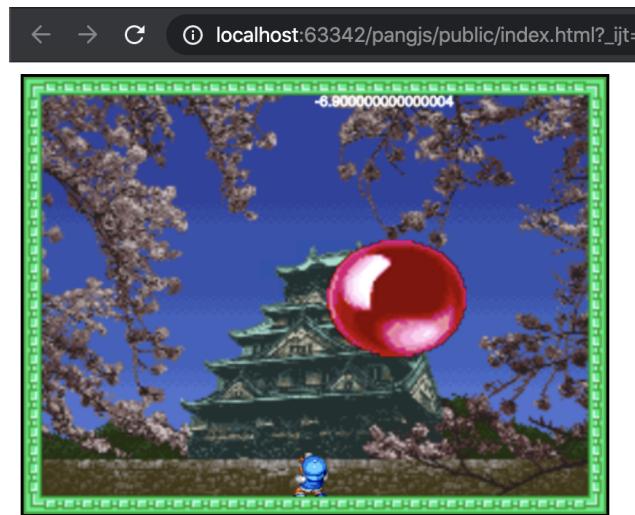


Figura 12: En la parte superior del canvas mostramos siempre, en blanco, el valor de `vy` tras el último rebote

Vemos que cuando si tras un rebote el valor absoluto de `vy` desciende de 5, la bola empieza a rebotar a una altura demasiado cercana a Buster. El truco es sencillo: comprueba que tras rebotar el valor absoluto de `vy` es siempre mayor que 5 y asígnale un -5 si esto no ocurriera⁹.

15. Gestión de niveles

Antes de empezar a gestionar niveles, vamos a añadir un atajo de teclado que nos permita pasar de nivel fácilmente. En concreto, vamos a darle a Buster un super-poder. Al pulsar la tecla k se ejecutará un método `killThemAll()` que llenará la pantalla de ganchos, separados por 5 pixels.

1. Añade en `input.js` un gestor para la tecla k.
2. Añade en `Player.js` el método `killThemAll`, que lanzará un gancho cada 5 pixels, empezando en la esquina izquierda, hasta la esquina derecha de la pantalla.
3. En `Settings.js`, permite lanzar -momentáneamente- hasta 100 ganchos. Recuerda desactivar este hack al terminar esta sección :)
4. Aprovechando que estás en `Player.js`, añade el método `setPos(pos)`, que recibe un array x,y y modifica la posición de Buster a la misma (necesitarás este método más adelante)

Necesitamos refactorizar `main.js` para dar soporte a la carga de niveles. En esta refactorización se ha introducido el método `startLevel` y se ha modificado el constructor de `loadBackground` (ahora toma como parámetro el nivel a mostrar).

⁸Cuando los valores de una variable cambian muy rápidamente, visualizarlos con `console.log` es insufrible

⁹Eres libre de ajustar más el salto mínimo si quieras incrementar la dificultad. Incluso podría ser una variable en función del nivel, o del radio de la bola...

La función startLevel toma como parámetros el nivel a cargar, el set de bolas, la factoría de bolas y el jugador. Debe llamar a loadLevel, cargar las bolas del JSON en balls y situar a Buster en la posición que indique el JSON.

Haz una copia de levels/1.json en levels/2.json (cambia la posición, número y color de las bolas).

Comprueba que al terminar de romper todas las bolas de nivel se pasa automáticamente al siguiente (recuerda que dispones de un super-arma en la tecla k).

16. Bonus

Un elemento que no puede faltar en un buen juego son los bonus. De vez en cuando, al explotar una bola, esta dejará caer un bonus que permanecerá en el suelo unos segundos. Si conseguimos atraparlo, Buster dispondrá de una nueva ayuda: doble gancho, gancho con anclaje, disparo, escudo, vida extra, congelar el tiempo, ... Aunque dejaremos preparado el juego para dar soporte a cualquiera de ellos, por ahora sólo implementaremos un único tipo de bonus, el gancho con anclaje.

- Añade estas tres variables a Settings:

```
static BONUS_DURATION = 5; // segundos que permanecerá
                           // en pantalla
static BONUS_SIZE = 30; // tamaño del Bonus
static BONUS_SPAWN_CHANCE = 0.15; // probabilidad de
                                  // obtener un bonus al romper una bola
```

- Programa el método update y draw de [Bonus.js](#).
- En main.js, carga la imagen con los sprites de los [bonus](#) (Fig. 13).



Figura 13: Sprites de los distintos tipos de bonus

- Añade un nuevo Set para guardar los bonus
- Siguiendo el mismo patrón que usamos para crear una factoría de bolas, crea una factoría de bonus (pásala como parámetro al constructor de CollisionManager). En loaders.js necesitarás añadir el método loadBonusFactory(bonusImage).
- en el loop principal, pinta (draw) y actualiza la posición de cada bonus
- En CollisionManager, dentro del método split_ball, llama al método spawn_bonus, pasándole como parámetro la posición de la bola
- El método spawn_bonus llama a bonusFactory¹⁰ para crear un nuevo bonus y lo añade al set de bonuses¹¹.

¹⁰Por ahora, sólo de tipo BonusType.chain_hook

¹¹Esto debe hacerse con una probabilidad de Settings.BONUS_SPAWN_CHANCE

- En el método check_collisions añade un bucle que recorra los bonus en pantalla, eliminando aquellos que estén marcados para ser eliminados ¹²

Por el momento, los bonus deben caer y quedarse unos segundos en el suelo antes de desaparecer. Sin embargo, Buster aún no puede recogerlos. Para dar soporte a la detección de una colisión entre el jugador y el bonus añde una nueva función box_to_box al fichero collisions.js.

Dentro de check_collisions en la zona de comprobación de los bonus, llama a la función box_to_box(bonus, this.player) para cada bonus. Si hay colisión, elimina el bonus del set de bonuses y llama a la función activate_bonus (por el momento, esta función sólo cambia el tipo de hook del player a **HookType.chain** ¹³).

17. Efectos de sonido

Añadirle efectos de sonido y música de fondo a ciertas pantallas o situaciones (por ejemplo, al comenzar una partida, al explotar una bola, al lanzar un gancho...) hace que el juego gane mucho en jugabilidad y atractivo.

Para añadir sonido a nuestro juego usaremos **Howler.js**, una librería JS para la gestión de audio en HTML5, potente y sencilla de utilizar.

Tal y como podemos ver en los ejemplos, cargar un archivo de audio y reproducirlo es tan sencillo como esto:

```
1 var sound = new Howl({
2   src: ['sound.mp3', 'sound.ogg']
3 }).play();
```

En este ejemplo se carga el fichero de sonido usando la clase Howl (en dos formatos, por defecto .mp3 y, si el navegador no lo soporta, .ogg) y se reproduce llamando al método play().

Existen muchos otros métodos y opciones (`fadeOut()`, `fadeIn()`, `autoplay`, `loop...`) muy interesantes para nuestro juego. Una de las funcionalidades más atractivas, ahora que dominamos el uso de sprites, es el de tratar un gran fichero de audio como si fuera una hoja de sprites. Es decir, podremos cargar en memoria un único fichero de audio (.mp3, .ogg, .wav...) que contenga distintas secuencias de sonido en el tiempo y recordarlas "para quedarnos con las que más nos interese. Como hemos dicho, el funcionamiento es exactamente igual que trabajar con sprites gráficos (de hecho, el atributo de Howl para gestionar este tipo de audios ¡se llama sprite!).

No obstante, en nuestro caso, disponemos de un fichero por cada efecto de sonido **sounds.zip**¹⁴.



Atención

En el navegador Chrome, por motivos de seguridad debes pulsar cualquier posición de pantalla antes de que el sonido empiece a sonar.

¹²Recuerda que sólo aguantarán en el suelo Settings.BONUS_DURATION segundos

¹³Debes añadir este atributo a Player y que el HookManager lo tenga en cuenta

¹⁴El juego original tenía más efectos de sonido... Para los más nostálgicos/as y/o detallistas, [aquí](#) puedes encontrar una buena colección

18. Game Over

En estos momentos, si nos has tocado el código más allá de lo que solicitado en el enunciado, Buster es inmortal. Sin embargo, disponemos de una clase CollisionManager que podemos aprovechar fácilmente para detectar colisiones entre bolas y personaje.

1. añade al constructor de CollisionManager una referencia al jugador (y pásaselo desde main.js)
2. en el método check_collisions, tras comprobar colisiones bola-hook, añade código para comprobar colisiones bola-jugador. En caso de colisión, muestra un mensaje por consola (o implementa la opción de añadir nuevo sprite para matar a Buster y quitarle una vida :)

19. Final

Si has llegado hasta aquí tienes ya una puntuación alta en la práctica y, por el camino, has aprendido técnicas de programación de videojuegos en HTML5 que te serán de utilidad en el futuro (no sólo para programar juegos, sino aplicaciones de todo tipo). Por lo tanto, ¡ enhorabuena ! Te mereces un descanso. Disfruta del [resultado final](#) y sal a celebrarlo :)

Cuando programas tu primer juego complejo, siempre te surgen nuevas ideas y opciones de mejora. De hecho, el juego de SuperPang original tiene más funcionalidades que no hemos programado en la práctica para evitar excedernos. Pero si quieres hacer que tu juego sea aún más entretenido, te propongo a continuación algunas funcionalidades que te ayudarán a lograrlo. No hay ayuda de código ni plantillas para implementar estas sugerencias pero merece la pena intentarlo, ¿ te animas a afrontar el reto ?



Atención

Las funcionalidades que se mencionan a continuación son totalmente OPCIONALES, pero si te animas a implementar alguna/s, podrás pasar de una nota muy buena a una nota excelente en la práctica :) Ojo, ¡ tampoco te excedes ! Te recomiendo comentarlo conmigo (email, tutorías, Telegram, Skype...) antes de implementar alguna de estas funciones extra. Recuerda que hay otras formas de conseguir puntos en DAWE, ¡ no tienen por qué gustarte los juegos !

19.1. Niveles

Añade nuevos niveles. Tienes los sprites de los mismos a tu disposición. Si implementas esta opción, deberías mostrar por pantalla el nivel actual en todo momento.

19.2. Tiempo

En el juego original hay un tiempo máximo en el que puedes completar el nivel. Aquí puedes ver una [partida completa](#).

19.3. Obstáculos

Algunas pantallas tienen escaleras a las que subirse. O puede que las bolas estén encerradas en un cubículo, cuyos pilares y suelo pueden romperse con un gancho.

19.4. Bonus

Hemos implementado el bonus de tipo gancho con anclaje, pero hay otros: reloj (para las bolas), vida extra, disparo....

19.5. Puntuación

Podrías mostrar una puntuación mientras el usuario está jugando. Dependiendo del tamaño de la bola que rompas obtendrás más o menos puntuación. Si implementas este apartado, también deberías mostrar el `High Score` (máxima puntuación obtenida por el mejor jugador hasta el momento) y las vidas que le quedan al jugador actual (en forma de pequeños gráficos de Buster o con simples números).

19.6. Hall of Fame

Puedes implementar el Hall of Fame (ranking de los 5 mejores jugadores) usando el API de `LocalStorage`. Al conseguir un récord, puedes pedirle al jugador su nombre en pantalla. El Hall of Fame se podrá consultar al final de cada partida (y/o al comienzo, tú eliges). En ese ranking se mostrará el nombre (7 caracteres), puntos y nivel alcanzados por los mejores 5 jugadores en orden decreciente. Un mismo jugador puede aparecer repetido varias veces con distintas puntuaciones. Mientras se muestra el Hall of Fame podrías hacer sonar de fondo algún efecto de sonido. La mejor puntuación del Hall of Fame es la que se muestra como High Score en el juego.

19.7. Soporte multibrowser

Es probable que el juego, tal cual está, sólo funcione en versiones modernas de Chrome y Firefox. Sería conveniente arreglarlo para que funcione también en navegadores para móviles, donde el jugador pudiera mover a Buster pulsando en las esquinas de la pantalla mediante gestión de `eventos touch`).

19.8. Soporte del GamePad API

HTML5 incorpora el `GamePad API` para poder usar mandos de consola conectados via USB. Puedes controlar a Buster usando uno de estos GamePads .