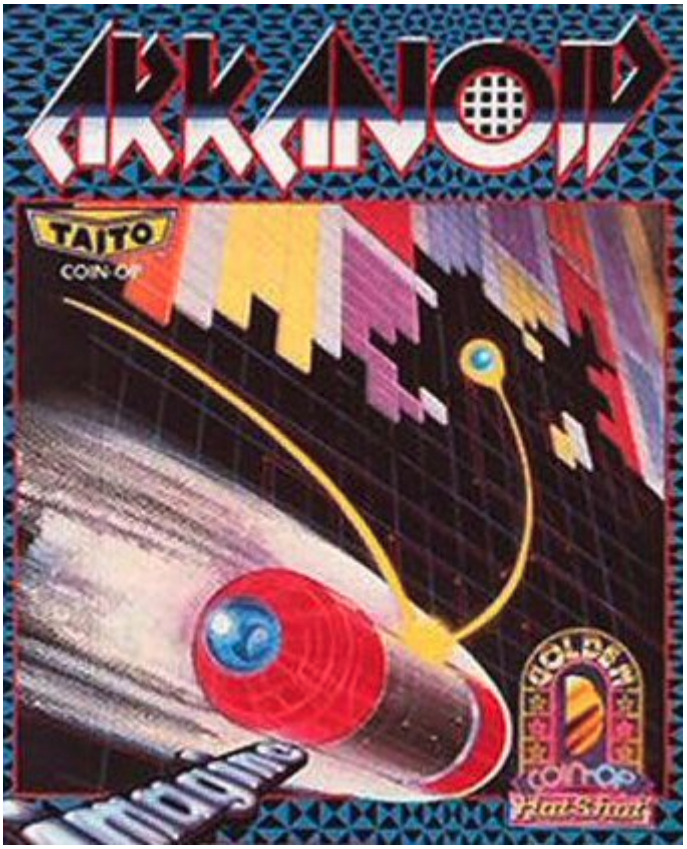


Autor: Juanan Pereira
Departamento: Lenguajes y Sistemas Inteligentes

Primera parte

1. Introducción y reglas básicas

Arkanoid es un videojuego de arcade desarrollado por Taito en 1986. Está basado en el juego Breakout de Atari de los años 70. Puedes ver un *longplay*[1] de este juego en YouTube.

El juego consiste en romper los rectángulos coloreados (ladrillos) que aparecen en la parte superior de la pantalla formando una pared o muro. El jugador dispone de tres *vidas* para finalizar el juego con éxito. Para romper los ladrillos, hay que hacer rebotar en ellos la bola, que aparece inicialmente en el centro de la pantalla, utilizando para ello el rectángulo que aparece en la parte inferior que hará las veces de raqueta (Fig. 1). En el juego original del Arkanoid, a esta raqueta se le conoce como la nave **Vaus**.

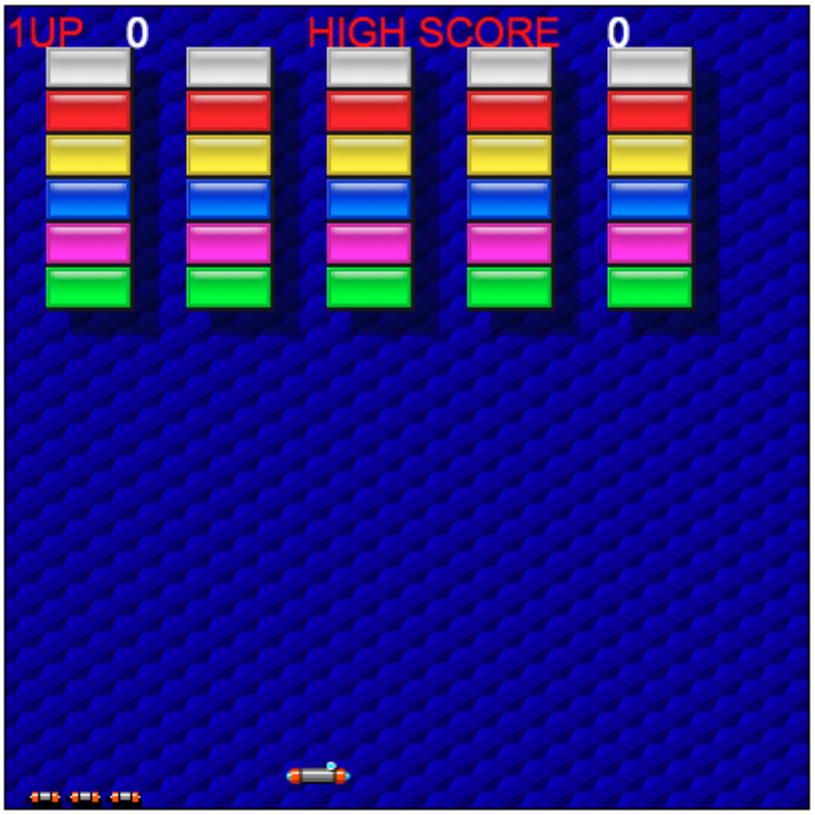


Figura 1: Pantalla inicial del Arkanoid

La raqueta siempre se mueve en horizontal y sólo en horizontal, siguiendo las órdenes del usuario. Los bordes de la pantalla son los bordes del tablero que la raqueta no puede sobrepasar.

La pelota puede rebotar tanto en la raqueta, como en los ladrillos y los bordes laterales. Los rebotes ocurrirán de acuerdo al principio físico de ``el ángulo de incidencia será igual al ángulo de reflexión'' que, tal y como se verá, es bastante sencillo de implementar. La Figura

[1] Grabación en vídeo realizada por un experto en el juego donde normalmente se ven todas las funcionalidades del mismo, incluida la última pantalla final.

2 muestra cuál será la trayectoria de la bola, tras rebotar contra la raqueta y posteriormente en la pared derecha. Tras rebotar contra la pared de ladrillos, la bola se moverá siguiendo la trayectoria indicada por la línea de puntos (esa línea se ha mostrado únicamente a título orientativo, no aparecerá en la pantalla de juego).

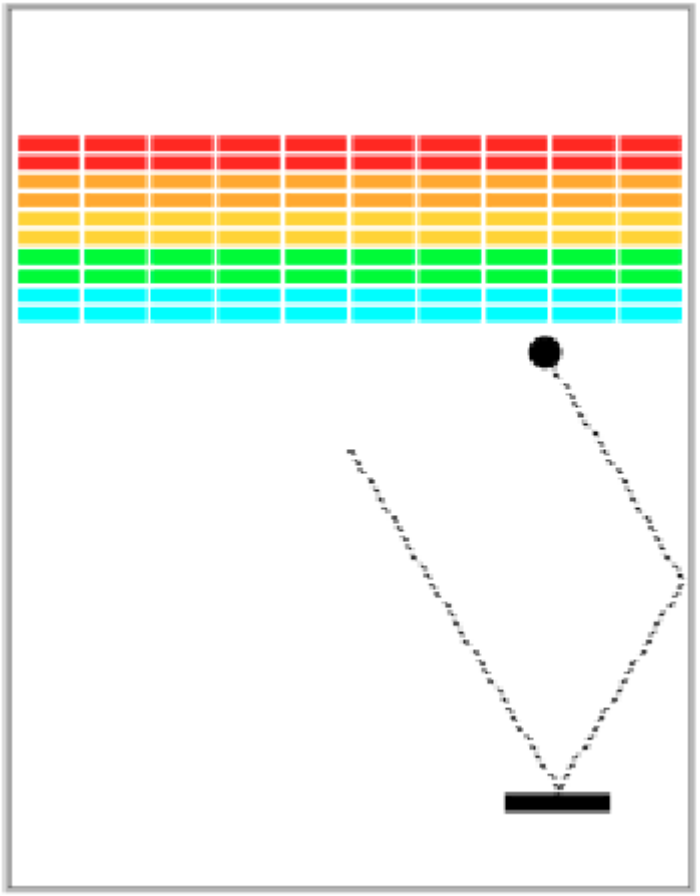


Figura 2: La bola puede rebotar contra Vaus o contra la pared.

En la misma figura se ve que la bola va a golpear un ladrillo de la primera fila del muro. Cuando esto ocurra, la bola rebotará, igual que en cualquier otra colisión, y el ladrillo golpeado desaparecerá[2].

Cuando la bola golpea el borde inferior de la pantalla (no ha sido golpeada por la raqueta), se pierde una vida. Cuando se haya destruido el último ladrillo rompible del muro, el jugador habrá superado el nivel y pasará a la siguiente fase o pantalla. En el Arkanoid original existen 33 niveles diferentes. El juego finaliza cuando se han perdido todas las vidas o se ha superado el último nivel.

[2] Asumimos que es un ladrillo normal. Como veremos más adelante, es posible implementar distintos tipos de ladrillos que requieran más de un golpe para ser eliminados o incluso ser irrompibles.

1.1. Interacción del usuario

El usuario puede usar las flechas del teclado (izquierda, derecha) para mover la raqueta. Si se implementa esta opción, el usuario también podrá pulsar la barra espaciadora, lo que le permitirá disparar a los ladrillos o sacar la bola al comienzo de la partida.

1.2. Panel de puntuación

Cada ladrillo roto incrementa la puntuación del usuario. Al igual que antes, esto puede ir en función del tipo de ladrillo que se haya roto, en caso de implementarlo.

El panel de puntuación, que está en la parte superior de la pantalla de juego, reflejará, aparte de los puntos acumulados, la puntuación máxima obtenida hasta el momento por cualquier jugador que hubiera jugado previamente al Arkanoid. En una primera versión, mostraremos también el número de vidas restantes. En la versión final, las vidas se mostrarán en la parte inferior, representadas gráficamente por medio de pequeñas naves Vaus (ver Fig. 1).

1.3. Niveles

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

1.4. Diseño de la práctica

Para ayudarte con el diseño de la práctica, dispondrás de plantillas Javascript de ayuda (JSFiddles) con las clases que tendrás que programar, junto con las definiciones de sus métodos (que tendrás que implementar). Iremos paso a paso, de forma incremental, probando en todo momento los métodos y clases que vayas implementando.

Fíjate que todos los métodos tienen un comentario 'AQUÍ TU CÓDIGO'. Debes borrar dicho comentario e implementar en su lugar cada método en cuestión. Para que la versión del Arkanoid que implementemos funcione, deberás desarrollar todos los métodos que se pedirán en el enunciado.

2. Esqueleto básico de un juego

Una forma sencilla de conseguir una animación en pantalla sería hacer uso del método *setTimeout*. Podemos ver un ejemplo aquí:

```
var addStarToTheBody = function(){
      document.body.innerHTML += "*";
      // calls again itself AFTER 200ms
      setTimeout(addStarToTheBody, 200);
};

// calls the function AFTER 200ms
```

```
setTimeout(addStarToTheBody, 200);
```

Esta forma de animación no es óptima. Si estamos tratando con un juego, como es el caso, *setTimeout* seguirá ejecutándose[3] aunque la pestaña del navegador no tenga el foco. Por otro lado, el navegador no sabe si esta ejecución cada intervalo de tiempo es para actualizar una animación en un canvas o para otro tipo de tarea totalmente diferente. Si supiera que el objetivo es realizar una animación, podría optimizar esa ejecución... y es aquí donde entra en juego el método *requestAnimationFrame*.

Este método funciona de forma parecida a *setTimeout* pero el navegador, sabiendo que es para hacer uso de una animación gráfica, intentará ejecutarlo a 60fps (60 frames por segundo, es decir, una vez cada 16,6 ms) de tal forma que dicha animación sea lo más fluida posible. Además, si la pestaña donde se está ejecutando la animación pierde el foco, *requestAnimationFrame* parará dicha animación hasta volver a recuperarlo (lo que permitirá liberar y optimizar recursos).

Podemos ver aquí un ejemplo de uso de *requestAnimationFrame*.

```
window.onload = function init() {
   requestAnimationFrame(mainloop);
};

function mainloop(timestamp) {
   document.body.innerHTML += "*";

   // call back itself every 60th of second
   requestAnimationFrame(mainloop);
}
```

Hay que tener en cuenta que el objetivo de 60fps puede no ser alcanzado. Depende mucho del dispositivo en el que se ejecute el código y la carga de trabajo que suponga la animación. En los ordenadores de escritorio de hoy en día no suele haber mayor problema para conseguir esta velocidad, pero en algunos dispositivos móviles podría no alcanzarse.

Este hecho representa un problema. Si animamos un objeto moviéndolo x posiciones por frame (x posiciones en cada vuelta del bucle) y asumimos 60fps, el objeto alcanzará cierta posición P en un segundo. Pero si cierto dispositivo no alcanza esa velocidad de refresco, el objeto no se desplazará a la misma posición P, sino que podría quedar bastante alejado de dicho punto P. Muchos juegos de acción solucionan este problema mediante una técnica conocida como animación basada en el tiempo (*time-based animation*). Veremos en una sección posterior (4) cómo funciona esta técnica.

[3] A mucha menor velocidad

Por el momento nos centraremos en encapsular el bucle de animación básica que hemos visto en el último ejemplo de código dentro de un objeto global GF (GameFramework)[4] . Ese objeto dispone de un método público \textit{start}. Al llamar a dicho método comenzará el bucle de animación principal (*mainloop*) que se ejecutará una vez por cada frame (generalmente, 60 veces en un segundo).

```
var GF = function(){
  var mainLoop = function(time){
      // Función Main, llamada en cada frame
      requestAnimationFrame(mainLoop);
  };
  var start = function(){
      requestAnimationFrame(mainLoop);
  };
  // Nuestro GameFramework sólo muestra una función pública al exterior
(el método start)
  return {
      start: start
  };
};
```

Con el esqueleto del juego anterior, podremos generar una nueva instancia de juego así:

```
var game = new GF();
// Lanzar el juego, comenzar el bucle de animación, etc.
game.start();
```

STOP

Tu primera tarea, sencilla, consiste en añadir el código necesario a *mainLoop* para que en cada frame se pinte un círculo de radio=5 píxels, borde negro, fondo rojo, en una posición aleatoria del canvas. Comprueba que tu aplicación pasa las pruebas unitarias del ejercicio: http://jsfiddle.net/f46aaocf/.

3. Midiendo los FPS

3.1. ¿A cuántos FPS se refresca la pantalla?

Como en todo buen juego, vamos a querer conocer la velocidad de refresco en pantalla que conseguimos en nuestro ordenador con el método *requestAnimationFrame*.

[4] Este objeto GF está basado en el ofrecido por el curso 'HTML5 Part 2: Advanced Techniques for Designing HTML5 Apps', recomendado en el foro https://www.edx.org/course/html5-part-2-advanced-techniques-w3cx-html5-2x-0

Para ello, debemos fijarnos primero en el parámetro *timestamp* que recibe *mainLoop*. Este parámetro describe el tiempo (en milisegundos) que transcurrió desde que comenzó la carga de la página con una precisión de microsegundos (varios decimales tras el milisegundo)[5]. ¿Y cómo podemos usar este valor para conocer los FPS de nuestro juego? Siguiendo estos tres pasos:

1. Contar el tiempo transcurrido sumando los deltas de tiempo de mainloop.
2. Si la suma de deltas es igual o superior a 1000, ha pasado 1 segundo.
3. Si, al mismo tiempo, contamos el número de frames que han sido dibujados (el número de veces que hemos entrado en mainloop), tendremos el valor FPS que buscamos (debería ser cercano a 60 frames/segundo).

```
// variables para contar frames/s, usadas por measureFPS
 var frameCount = 0;
 var lastTime;
 var fpsContainer;
 var fps;
 var measureFPS = function(newTime){
   // la primera ejecución tiene una condición especial
   if(lastTime === undefined) {
      lastTime = newTime;
      return;
   }
   // calcular el delta entre el frame actual y el anterior
   var diffTime = newTime - lastTime;
   if (diffTime >= 1000) {
      fps = frameCount;
      frameCount = 0;
      lastTime = newTime;
   }
   // mostrar los FPS en una capa del documento
   // que hemos construído en la función start()
   fpsContainer.innerHTML = 'FPS: ' + fps;
   frameCount++;
 };
```

Llamaremos a la función *measureFPS* desde mainloop:

```
var mainLoop = function(time){
      measureFPS(time);

  // llamar a mainloop cada 1/60 segundos
```

[5] Realmente es el mismo valor que el que obtendríamos como resultado al llamar a performance.now(). Más info en https://developers.google.com/web/updates/2012/08/When-milliseconds-are-not-enough-performance-now y https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

```
  requestAnimationFrame(mainLoop);
};
```

STOP

Modifica el método drawVaus() para pintar la raqueta Vaus en pantalla en la posición x,y que llega como parámetro. Vaus tendrá por el momento un color negro, altura 10 y anchura 30. Para hacer más sencillo el ejercicio, te vendrá bien usar los métodos save() y restore() del contexto del canvas. Comprueba que tu aplicación pasa las pruebas unitarias del ejercicio: http://jsfiddle.net/juanan/e5g6wp5q/. Verifica que en tu ordenador de mesa consigues unos 60fps.

3.2. Mover Vaus a izquierda y derecha

Nuestro siguiente objetivo será mover Vaus continuamente, dentro del *mainloop*, a izquierda -hasta que toque la pared izquierda- y derecha -hasta que toque la pared derecha. Un toque en una pared hará que Vaus "rebote'" al lado contrario (por el momento, más adelante modificaremos este comportamiento).

STOP

Implementa el método updatePaddlePosition que mueve a Vaus de izquierda a derecha según se ha indicado. Comprueba que tu aplicación pasa las pruebas unitarias del ejercicio: http://jsfiddle.net/juanan/ttwc6w1a/.

4. Animación basada en tiempo (Time-based animation)

Tal y como se ha implementado el movimiento de Vaus hasta ahora, la velocidad de la raqueta depende de los FPS que soporte nuestro dispositivo. En un ordenador de escritorio estaremos desplazándonos 5px en cada frame. Si la velocidad de refresco es de 60 fps, en un segundo nos habremos desplazado 300px. Sin embargo, en un dispositivo que sólo alcance 50 fps, sólo nos habremos desplazado 250px. Hay que evitar en lo posible que la velocidad de los objetos del juego dependa de los FPS que soporte el dispositivo.

Afortunadamente hay una solución sencilla para este problema: aplicar la técnica de animación basada en tiempo (no en fps). En nuestro juego, basta con fijar la velocidad en píxeles/segundo a la que queremos que se mueva Vaus (en nuestro ejemplo 300px) y ajustar la velocidad a la que se debe mover en cada frame para alcanzar ese objetivo.

Por ejemplo, a 60fps, sabemos que cada 1/60 segundos (delta = 16.66 ms) se actualizará la posición de Vaus. El número de pixels a mover será:

```
  num_pixeles_frame = (velocidad en px/seg * 16.66 ms) / 1000
```

En nuestro ejemplo:

```
  num_pixeles_frame = (300 px/seg * 16.66ms) / 1000 = 4.998 (es decir, 5
px)
```

Si la velocidad de refresco fuera de 50fps, el valor de delta será 1/50 = 20ms. Entonces, ¿cuántos pixels deberíamos avanzar la raqueta en cada frame para mantener una velocidad de 300px/seg?

```
num_pixeles_frame = (300 px/seg * 20ms) / 1000 = 6 pixels
```

STOP

Implementa el método calcDistanceToMove que tomando como parámetros dos valores (delta y velocidad) calcule el número de píxeles que debe moverse un objeto en cada frame para alcanzar esa velocidad en píxeles/seg. Actualiza el método updatePaddlePosition para que haga uso de calcDistanceToMove. Comprueba que tu aplicación pasa las pruebas unitarias del ejercicio: http://jsfiddle.net/juanan/3rt78ujx/7 .

5. Eventos de teclado

Para poder mover la raqueta Vaus con las flechas 'Izquierda' y 'Derecha' , al mismo tiempo que, por ejemplo, disparamos con la barra espaciadora, debemos añadir código para gestionar los eventos de teclado, es decir, conocer cuándo el usuario ha pulsado una tecla. Hay que tener en cuenta que pudiera ser que hubiera dos teclas pulsadas a la vez (izquierda/derecha + espacio). Para ello definiremos un objeto llamado *inputStates* con los atributos *left*, *right* y *space*. Estos atributos tomarán valores booleanos en función de si la tecla correspondiente está pulsada o no. Ahora el método *updatePaddlePosition* -que se ejecuta en el mainloop- incluirá una comprobación del estado de *inputStates* para saber si debe mover Vaus a izquierda o derecha (el disparo lo dejaremos para más adelante).

Debes añadir un gestor de pulsaciones de tecla (*eventListener*) al método *start*. Este gestor actualizará el valor de los atributos *left*, *right* y *space* del objeto *inputStates* dependiendo de si el usuario ha pulsado o soltado la tecla correspondiente.

ATENCIÓN

No hay pruebas unitarias para este ejercicio, pero no pases al siguiente hasta no terminar este. Puedes usar este código como plantilla: http://jsfiddle.net/juanan/7fx62awb/. Recuerda actualizar también el método \textit{updatePaddlePosition} para que Vaus no vaya de izquierda a derecha automáticamente, sino que haga caso del valor de los atributos left, right, space de

inputStates. Nota: Cuando el usuario pulse espacio basta con que muestres por consola un mensaje.

6. Gestión de las pelotas

Con perdón ;)

Nuestro siguiente objetivo será crear una bola que rebote contra las paredes. Hay que tener en cuenta que en el juego puede llegar a haber más de una bola en movimiento (al capturar un bonus verde) por lo que haremos uso de una clase (función constructora) que nos permita crear tantas bolas como queramos.

El constructor de la clase *Ball* debe tomar como parámetros la posición inicial (x,y), el ángulo inicial de salida, la velocidad inicial, diámetro de la bola y un atributo sticky que indica si la bola está pegada a la raqueta[6]. Esta clase tendrá dos métodos: draw() y move(), que serán ejecutados en cada frame.

- El método move() moverá la bola en incrementos x,y calculados de la siguiente forma:

```
incX = this.speed * Math.cos(this.angle);
incY = this.speed * Math.sin(this.angle);
```

  Una vez calculados incX e incY recuerda aplicar a esos valores una animación basada en tiempo antes de actualizar la posición x,y de la bola.

- El método draw() pinta la bola en la posición actual x,y.

STOP

Implementa los métodos que se han indicado y comprueba que tu aplicación pasa las pruebas unitarias del ejercicio: http://jsfiddle.net/juanan/tmws5vzm/7/. Inicialmente, la bola saldrá de la posición (10,70), con un ángulo de PI/3, una velocidad inicial de 10 px/seg, un radio de 6 pixels y no será sticky (ya gestionaremos esto más adelante). Importante: haz que la bola no salga del techo de la pantalla (puede salirse por los bordes laterales).

[6] Cuando comienza un juego en el juego Arkanoid original, la bola está pegada a Vaus. También hay un bonus que permite hacer la raqueta sticky. Por ahora simplemente dejaremos el atributo preparado.

ATENCIÓN

Observa que en la definición de variables de GF hemos añadido el siguiente código para gestionar varias bolas a la vez en el juego (inicialmente partimos sólo con una).

```
var balls = [];
```

Para este test, debes introducir en ese array una bola (que previamente habrás instanciando con los parámetros que pide el ejercicio).

La bolas del array se actualizan en cada frame por medio de una llamada a la función updateBalls -ya implementada- que lo único que hace es llamar al método move() y draw() de cada bola.

6.1. Crear bolas que reboten en las paredes

Implementa el método testCollisionWithWalls() que toma como parámetros la bola actual, la anchura y la altura del tablero y cambia la trayectoria de la bola en caso de que ésta toque alguno de los bordes. Devuelve true si el borde que la bola ha tocado es el inferior, false en cualquier otro caso.

Ten en cuenta que la posición x,y se refiere al centro de la bola...

Los rebotes siguen la siguiente fórmula en caso de tocar las paredes laterales:

```
ball.angle = -ball.angle + Math.PI;
```

y ésta otra en caso de tocar las paredes inferior o superior:

```
ball.angle = -ball.angle;
```

STOP

Implementa el método indicado y comprueba que tu aplicación pasa las pruebas unitarias del ejercicio: http://jsfiddle.net/o9ny1qzj/

6.2. Detectar colisión con raqueta

Modifica el método updateBalls() para que tenga en cuenta la posible colisión con la raqueta. Para ello, puedes llamar a la función circRectsOverlap que comprueba si existe colisión entre un rectángulo (con vértice superior izquierdo en x0,y0 de anchura w0 y altura h0) y un círculo (con centro en cx,cy y radio r).

```
// Collisions between rectangle and circle
```

```
    function circRectsOverlap(x0, y0, w0, h0, cx, cy, r) {
        var testX = cx;
        var testY = cy;

        if (testX < x0)
            testX = x0;
        if (testX > (x0 + w0))
            testX = (x0 + w0);
        if (testY < y0)
            testY = y0;
        if (testY > (y0 + h0))
            testY = (y0 + h0);
        return (((cx - testX) * (cx - testX) + (cy - testY) * (cy -
testY)) < r * r);
    }
```

En caso de colisión entre la bola y Vaus, modifica el ángulo de la bola tal y como has realizado en el ejercicio anterior.

> **ATENCIÓN**
>
> No hay pruebas unitarias para este ejercicio, pero no pases al siguiente hasta no terminar este. Nota: no te conformes con hacer rebotar la pelota con la raqueta, intenta que la pelota no entre "dentro" de la raqueta (o si lo hace, ten en cuenta que antes de hacerla rebotar, deberías actualizar su posición y). Si no lo haces, es posible que la pelota se quede atrapada dentro de la raqueta en un bucle infinito.

7. Construyendo paredes

Vamos a construir una pequeña pared de ladrillos (sólo una decena para estas pruebas). Para ello, prepararemos una clase Brick (con un constructor de 3 parámetros, x,y, color), un método draw() y un array de ladrillos. Este array resumirá los datos que necesitamos para instanciar todos los ladrillos de la pared. En el método start de nuestro GF (GameFramework) llamaremos al método createBricks() que instanciará todos los ladrillos y los incluirá en un array de objetos llamado bricks.

```
var ladrillos = [
      // grey
      {x:20,y:20,c:'grey'},
{x:(20*2+ANCHURA_LADRILLO),y:20,c:'grey'},{x:20*3+ANCHURA_LADRILLO*2,y:2
0,c:'grey'},{x:20*4+ANCHURA_LADRILLO*3,y:20,c:'grey'},
{x:20*5+ANCHURA_LADRILLO*4,y:20,c:'grey'} ,
      // red
      {x:20,y:42,c:'red'},
```

```
{x:20*2+ANCHURA_LADRILLO,y:42,c:'red'},{x:20*3+ANCHURA_LADRILLO*2,y:42,c
:'red'},{x:20*4+ANCHURA_LADRILLO*3,y:42,c:'red'},
{x:20*5+ANCHURA_LADRILLO*4,y:42,c:'red'}];
```

Dentro de mainloop llamaremos al método drawBricks en cada frame. Este método se limitará a pintar todos los ladrillos del array bricks en pantalla.

STOP

Implementa los métodos que se han indicado y comprueba que tu aplicación pasa las pruebas unitarias del ejercicio: http://jsfiddle.net/uvxdb6pb/.

7.1. Rompiendo ladrillos

El método updateBalls() llamará internamente al método testBrickCollision(ball) que toma como parámetro la bola actual y comprueba si esta bola colisiona con la posición de alguno de los ladrillos. Para ello, el método testBrickCollision() hace uso del método intersects(). Este método no sólo comprueba si una bola ha colisionado contra un ladrillo en concreto sino que además, devuelve un objeto literal con varios atributos:

- position: lado del ladrillo sobre el que se ha producido la colisión (left, right, top, bottom).
- c: valor booleano que indica si ha habido colisión o no.

En caso de colisión de la bola con un ladrillo, modificaremos el ángulo de la bola adecuadamente y eliminaremos el ladrillo del array de ladrillos. Para darle más complejidad al juego, cada vez que se elimine un ladrillo aumentaremos la velocidad de la bola actual. Finalmente, el método testBricksCollision() devuelve el número de ladrillos restantes.

STOP

Crea una variable bricksLeft que llevará el número restante de ladrillos en pantalla. Actualiza esta variable dentro del método createBricks. Termina de desarrollar el código que se ha indicado (método testBrickCollision) y comprueba que tu aplicación pasa las pruebas unitarias del ejercicio: http://jsfiddle.net/tmocn6Lv/.

8. Control de las Vidas

Debemos gestionar las vidas del jugador. Inicialmente partiremos con 3 vidas y cada vez que la pelota toque la parte inferior de la pantalla perderemos una. Si llegamos a cero vidas se acabará el juego (Game Over). Crea una nueva variable llamada *lifes* e inicialízala a 3. A continuación, modifica el método updateBalls() para que en caso de que la pelota toque la pared inferior, se elimine una bola de la pantalla. Si era la última bola en pantalla, resta una vida al usuario.

STOP

Termina de desarrollar el código que se ha indicado y comprueba que tu aplicación pasa las pruebas unitarias del ejercicio: http://jsfiddle.net/5y9a53oy/. Añade también un método displayLifes() al mainloop para que muestre en la esquina superior derecha el número de vidas actual en todo momento.

8.1. Primera refactorización

Cuando el jugador pierde una vida (pero aún le queda alguna para seguir jugando), debemos reanudar el juego desde ese punto. Si no lo hacemos, la bola se perderá por la parte inferior y no saldrá una nueva. Debemos marcar de alguna forma esta situación, es decir, marcar que el jugador ha perdido una vida pero que puede seguir jugando, por lo que debemos lanzar una nueva bola al canvas. Para conseguir esto, vamos a refactorizar la parte del código que gestiona a Vaus. Para ello, crearemos un objeto literal llamado paddle (raqueta):

```
// Vaus se gestionará con este objeto
  var paddle = {
      dead: false,
      x: 10,
      y: 130,
      width: 32,
      height: 8,
      speed: 300,
       sticky: false
  };
```

Fíjate que hemos añadido dos nuevos atributos:

- dead: indica si acabamos de perder una vida (y por tanto, el juego debe lanzar una nueva bola si es que aún nos quedan vidas).
- sticky: nos servirá para saber si la raqueta es pegajosa (la bola se debe quedar pegada a la raqueta).

STOP

Añade el objeto paddle al juego y comenta la declaración de las variables vausHeight, vausWidth, speed, x e y. Intenta ejecutar el juego. Verás que en la consola se mostrarán errores porque ya no existen las variables que acabas de comentar. Sustituye en el código cada variable que falta por su atributo equivalente. Por ejemplo, sustituye speed por paddle.speed, vausWidth por paddle.width, etc. Asegúrate de que tras la refactorización el juego sigue funcionando como antes, sin errores, y que al

perder una vida el valor de *paddle.dead* es true. Comprueba que tu código pasa las pruebas unitarias: http://jsfiddle.net/qwdcr78m/.

STOP

Añade el código necesario a *mainloop* para que, cada vez que se pierda una vida, se genere una nueva bola inicial y se pueda seguir jugando. Este ejercicio no tiene pruebas unitarias, pero asegúrate de que funciona tal y como se indica en el enunciado.

8.2. Game Over

En el ejercicio anterior te habrás dado cuenta de un detalle: si pierdes 3 vidas, el juego sigue (pudiendo incluso seguir jugando con un número de vidas negativo). Tenemos que arreglar esto. Es decir, si el número de vidas que quedan es 0, debemos mostrarle al jugador un mensaje de *Game Over* y dejar de actualizar la posición de las bolas, raqueta y estado de los ladrillos en pantalla. Para ello, vamos a crear una nueva variable *currentGameState* que nos permita saber en qué estado del juego nos encontramos: jugando (gameState.gameRunning) o con la partida finalizada (gameState.gameOver). Inicialmente currentGameState se inicializa a gameState.gameRunning.

STOP

Modifica *mainloop* para que se pregunte por el valor de *currentGameState*. En caso de que se esté jugando (hay vidas), todo sigue igual. En caso de que hayamos perdido todas las vidas, se cambiará el estado de forma correspondiente y se mostrará en pantalla el mensaje *Game Over* en el medio de la pantalla (con letras blancas sobre un canvas con fondo totalmente negro). Comprueba que tu código pasa las pruebas unitarias: http://jsfiddle.net/pwap8xea/. Nota: fíjate que para hacer este test rápidamente, hemos cambiado el número de vidas inicial a 1.

¡ENHORABUENA! Has terminado de programar tu propio juego del Arkanoid :-) 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 como el de la figura 1 (y obtener una buena nota en la práctica :) te animo a que sigas trabajando un poco más.

La siguiente parte de la práctica no dispondrá de tests unitarios. 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...

Segunda parte

9. Refactorización

Vamos a realizar una refactorización del código. La primera a nivel local y la segunda a nivel global. Empecemos por la refactorización local.

> **STOP**
>
> La función start() es demasiado larga y compleja. En este ejercicio, debes extraer los gestores de teclado a una función de nombre inicializarGestorTeclado().

Una vez realizado lo anterior, pasemos a una refactorización de mayor calado. Consistirá en separar las distintas funcionalidades en ficheros JS.

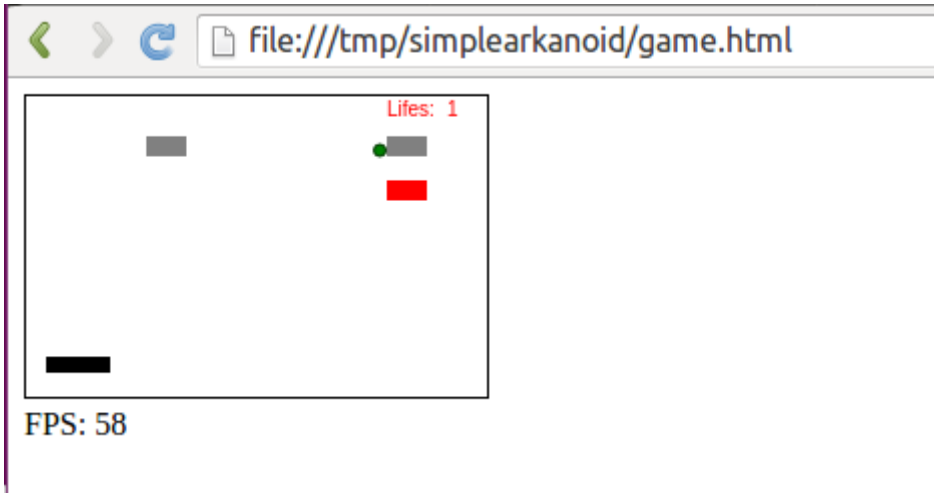
En concreto, elimina el código de test y copia el código JavaScript del último JSFiddle a un fichero externo js/game.js. A continuación, crea un fichero llamado css/game.css con la hoja de estilo copiada de JSFiddle. Finalmente, crea un fichero externo llamado game.html con el siguiente código:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>ARKANOID</title>
  <link rel="stylesheet" href="css/game.css">
  <script src="js/game.js"></script>
</head>
<body>
    <canvas id="canvas"  width="230" height="150"></canvas>
</body>
</html>
```

Como ves, lo único que hemos hecho ha sido extraer el código de JSFiddle a ficheros locales. Prueba a cargar game.html y asegúrate de que todo sigue funcionando como antes.

> **STOP**
>
> Ten cuidado con la ejecución de las funciones JavaScript. Asegúrate de que el código se ejecute tras el evento *onload*, es decir, tras haber cargado todos los recursos (html, js, css). Si no lo haces así, obtendrás errores de ejecución.

El resultado final debería ser similar a este:



10. Pintando sprites

La raqueta Vaus, los ladrillos, la bola, el fondo de pantalla y el texto con el número de vidas restantes son demasiado sencillos. Los juegos de hoy en día usan sprites para mejorar la apariencia y jugabilidad. En esta sección aprenderemos a trabajar con sprites. Para ello, haremos uso de dos librerías JavaScript (guárdalas en la carpeta js/ de tu proyecto):

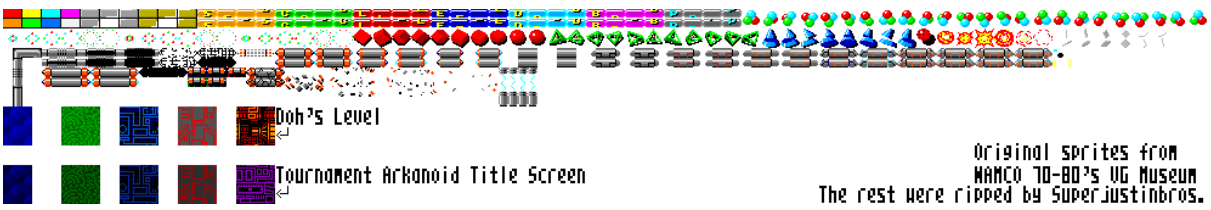
- Pequeña librería para manejo de sprites, sprites.js.
- Librería para carga de recursos externos (audio, gráficos), resources.js

No te olvides de cargar estas dos librerías desde tu código HTML.

El uso de sprites nos va a permitir gestionar animaciones gráficas de alta calidad en nuestro juego. La técnica es sencilla y se puede dividir en estos tres pasos:

1. Cargar la hoja con los sprites en memoria.
2. Obtener la posición del sprite que se quiera usar.
3. Llamar periódicamente a una función para actualizar la posición y redibujar el frame del sprite que se quiere animar.

Veamos estos pasos uno a uno. A modo de ejemplo, haremos algo sencillo: cargar un sprite con Vaus, que consta únicamente de una animación con dos frames (enmarcados en rojo en la siguiente imagen):



Puedes descargar la imagen desde aquí. Ahora hay que cargar esta imagen en memoria y extraer los sprites que necesitemos. Veamos cómo hacerlo.

10.1. Cargar sprites en memoria

¿Por qué cargamos una hoja con todos los sprites en lugar de una imagen por cada sprite? Cargar una hoja con todas las imágenes (sprites) es mucho más rápido que cargar cientos de pequeños ficheros (uno con cada sprite). Por otro lado, existen webs con las hojas de gráficos de los juegos clásicos (por ejemplo, ésta de aficionados a los Sprites, que he usado como patrón para los ladrillos de la figura 1).

Para cargar la hoja de sprites haremos uso de la librería resources.js que hemos obtenido en el punto anterior:

```
resources.load([
            'img/sprites.png'
      ]);
resources.onReady(init);
```

En concreto, el método *load* nos permite cargar cualquier recurso en memoria, y avisarnos cuando la carga haya finalizado mediante la emisión de un evento *onReady*.

En el ejemplo, cargamos el fichero *img/sprites.png* y cuando detectamos que está disponible en memoria, respondemos al evento *onReady* llamando a la función *init*[7]. Esta llamará, a su vez, a la función startNewGame() para inicializar las variables del juego y comenzar el loop principal. En resumen:

```
 function init(){
      startNewGame();
      // comenzar la animación
      requestAnimationFrame(mainLoop);
}

function startNewGame(){
      balls.push(new Ball(10, 70, Math.PI / 3, 100, 6, false));
      createBricks();
}

var start = function() {
      // capa div para visualizar los fps
      fpsContainer = document.createElement('div');
      document.body.appendChild(fpsContainer);

      inicializarGestorTeclado();
```

[7] Este tipo de funciones se denominan funciones callback.

```
    resources.load([
          'img/sprites.png'
       ]);
     resources.onReady(init);
};
```

10.2. Obtener la posición del sprite

El sprite que queremos usar para Vaus está situado en el punto 224,40 de la hoja de sprites. Cada frame (hay dos), tiene una anchura de 32 y una altura de 8 píxeles[8]. Nos gustaría que la animación entre frames se realizara X veces por segundo[9], primero el frame 0, luego el 1, luego el 0, luego el 1, etc. Esta configuración se consigue con la siguiente línea (un nuevo atributo que añadiremos al objeto paddle):

```
sprite: new Sprite('img/sprites.png', [224,40], [32,8], 16, [0,1])
```

10.3. Actualizar la posición del sprite

La nave Vaus se mueve con el teclado y su posición se recalcula mediante la función updatePaddlePosition() que ya tenemos programada. Lo único que haremos será añadir a esa función, como primera instrucción, la siguiente llamada:

```
paddle.sprite.update(delta);
```

lo que permite calcular qué frame del sprite de Vaus hay que dibujar (recordemos, el 0 o el 1, 16 veces por segundo).

10.4. Redibujar el sprite

Una vez que sabemos qué frame de Vaus hay que dibujar, sólo nos queda pintar dicho sprite. Para ello, modificaremos la función \texttt{drawVaus()} para que en lugar de pintar un rectángulo negro, pinte el sprite correspondiente:

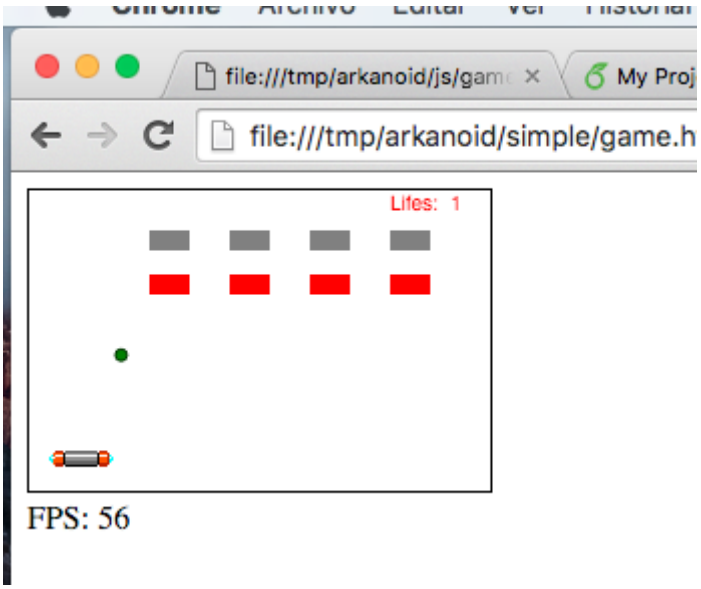
```
paddle.sprite.render(ctx);
```

Si programas todo como se ha indicado, deberías poder jugar al Arkanoid con una bonita raqueta Vaus en forma de sprite (los bordes azules del sprite deben parpadear).

---

[8] Ayúdate de esta web para calcular posiciones de sprites https://getspritexy.netlify.app/.

[9] Realmente esta velocidad (16 en el ejemplo) sirve para calcular el siguiente índice del frame de la animación, así: indice_frame = velocidad*delta , donde delta es el tiempo que pasa entre cada vuelta del \texttt{mainloop}. Tendrás que jugar con este valor para ajustarlo a tu gusto.



11. Un fondo de pantalla bonito

El fondo blanco actual no es nada atractivo. Solucionemos esto con un poco de magia HTML5. Lo primero que tenemos que recordar es que el API de manipulación del <canvas> dispone de un método *fillStyle* que toma como parámetro un patrón (generado a través del método *createPattern*[10]) para rellenar con el mismo el fondo del canvas. Es curioso, porque el método *createPattern* necesita como parámetro la imagen que servirá como patrón...¡dentro de un canvas!. Afortunadamente, gracias a la librería sprites.js que ya estamos usando, no tendremos que programar nada nuevo. ¿Por qué? Porque si nos fijamos en la clase Sprite, veremos que disponemos de un método image(), que nos devuelve un canvas con la imagen del sprite dentro. Por tanto, podremos cargar el fondo como un sprite (sin frames de animación, o lo que es lo mismo, con un único frame) y llamar a la función:

```
terrainPattern =  ctx.createPattern ( terrain.image(), 'repeat') ;  //
repeat forma un mosaico con el fondo
```

Donde terrain es una variable de tipo Sprite[11] que guarda una referencia al fondo:

```
terrain = new Sprite('img/sprites.png', [posX,posY], [anchura, altura]);
```

Deberás determinar la posición (x,y) donde se sitúa el patrón de fondo que quieras usar (así como la anchura y altura de ese patrón).

Añade esas dos líneas a una función initTerrain() a la que deberás llamar al comienzo del método startNewGame().

---

[10] Ver https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/fillStyle.

[11] Fíjate que sólo los primeros tres parámetros este constructor son obligatorios.

Si realizas todo correctamente verás algo como la imagen de la izquierda. Si fallas en el cálculo de las posiciones, verás algo como la imagen de la derecha.

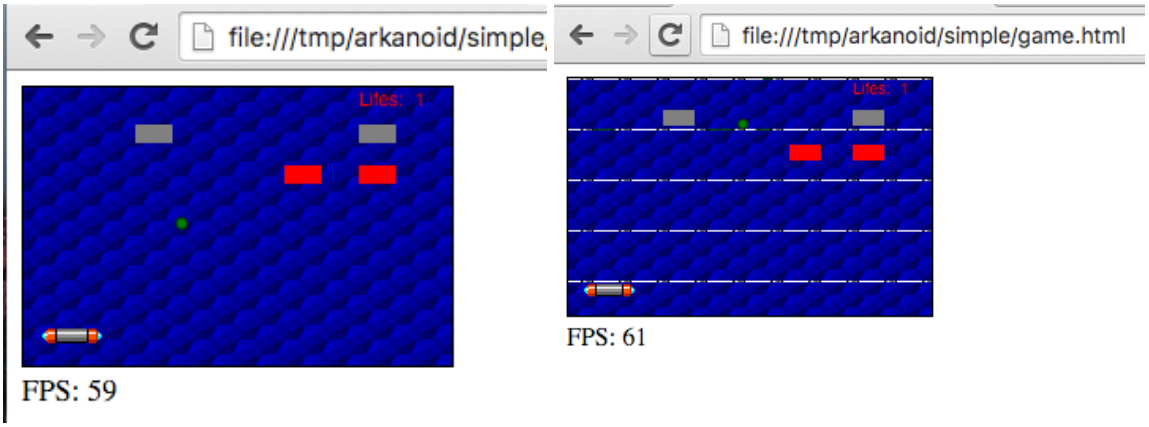


Figura 3: A la izquierda, ejemplo de fondo correctamente calculado. A la derecha, pueden apreciarse imprecisiones en el cálculo del fondo.

12. Convertir ladrillos a sprites

En la misma hoja de sprites (*sprites.png*) podemos ver que hay una serie de imágenes preparadas para colorear los ladrillos. Vamos a usarlas para darle un toque de color a nuestra pared de ladrillos. Para ello, bastará con añadir un atributo sprite a la clase Brick:

```
        this.sprite = new Sprite('img/sprites.png', coords[color], [w,h]);
```

Fíjate que estamos usando el mismo constructor (exactamente los mismos parámetros), pero le hemos añadido un nuevo atributo (sprite). Este atributo obtendrá de un array asociativo (coords) las coordenadas del ladrillo en cuestión (las coordenadas del ladrillo que nos interese dentro de la imagen sprites.png) dependiendo, lógicamente, del color que llegue como parámetro. w y h son dos constantes que indican la anchura y altura que queremos para nuestros ladrillos.

ATENCIÓN

Piensa que la función *clearCanvas* se ejecuta dentro del *mainLoop*... Tendrás que cambiarla para que no se limite a limpiar la pantalla con un color blanco, sino que debe rellenar el fondo con el patrón preparado en *terrainPattern*.

La función draw() de clase Brick simplemente se posicionará en la coordenada x,y[12] y pintará el sprite:

[12] Seguramente te venga bien hacer los tests del tema canvas disponibles en el bot de Telegram para saber cómo hacer esto.

```
      this.sprite.render(ctx);
```

Recomendación: para mayor legibilidad, extrae el código de Bricks a un fichero JS (por ejemplo, bricks.js) y cárgalo desde game.html.

13. Efectos de sonido

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

Para añadir sonido al juego usaremos Howler.js, una librería JS para la gestión de audio en HTML5, potente y sencilla de utilizar. Descárgala aquí.

Tal y como podemos ver en los ejemplos de la web de Howler, cargar un archivo de audio y reproducirlo es tan sencillo como ésto:

```
var sound = new Howl({
  urls: ['sound.mp3', 'sound.ogg']
}).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 "recortarlas" 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!).

Veamos un ejemplo. Supongamos que el fichero sounds.mp3 contiene tres secuencias de sonido que nos interesa usar: blast (del segundo 0 al 2, láser (del segundo 3 al 3.7) y winner (del segundo 5 al 9). Entonces lo cargaríamos así:

```
var sound = new Howl({
  urls: ['sounds.mp3', 'sounds.ogg'],
  sprite: {
    blast: [0, 2000],
    laser: [3000, 700],
    winner: [5000, 9000]
  }
```

```
});

// Si quisiéramos hacer sonar el láser:
sound.play('laser');
```

13.1. Audio inicial

Vamos a usar Howler para cargar varios ficheros de audio. Cargar un audio en memoria seguramente le lleve un tiempo al navegador (especialmente si el fichero físico se encuentra en un servidor remoto[13]). Afortunadamente Howler permite gestionar el evento \texttt{onload} que se elevará al terminar de cargar cada fichero. Por ejemplo, para cargar la música que queremos que suene al comienzo de la partida (cuando sacamos la bola), haremos lo siguiente:

```
function loadAssets(callback) {
          // Cargar sonido asíncronamente usando howler.js
          music = new Howl({
                urls: ['http://localhost/assets/Game_Start.ogg'],
              volume: 1,
              onload: function() {
            callback();
          }
         }); // new Howl
}
```

ATENCIÓN

Fíjate que la URL sigue el protocolo HTTP. Si quieres que esto funcione, deberás tener el juego en el mismo servidor desde el que haces la llamada. Recuerda también que debes cargar en game.html la librería howler.js.

STOP

Deberás modificar tu código ligeramente para llamar al método loadAssets() en el momento y forma precisas. Sugerencia: dentro del método init(). Piensa cómo hacer para que las líneas que ejecutaba init() (startNewGame() y requestAnimationFrame()) las ejecute TRAS el loadAssets().

¡No te olvides de hacer sonar el sonido allá donde lo necesites! (music.play()).

[13] El fichero de audio Game_Start.ogg al que se hace referencia puedes descargarlo desde aquí.

13.2. Otros efectos de audio

El fichero sounds.mp3 contiene múltiples efectos de audio usados en el juego del Arkanoid. deberás usar este fichero como un sprite de sonidos. Como ayuda, aquí puedes ver un extracto de la función loadAssets() modificada para que cargue también sounds.mp3 (y algunos de los sprites de audio que contiene).

```
....
onload: function(){
     sound = new Howl({
         urls: ['http://localhost/assets/sounds.mp3'],
         volume: 1,
         sprite : {
              point: [0,700],
              salir: [1000,1700],
              empezar: [3000,2700]
                  ...
```

STOP

Elige alguno de los sprites de audio para que suene cuando Vaus golpea la bola y otro sprite de audio para que suene al golpear un ladrillo.

14. Rebote con efecto - Spin effect

Tal y como hemos implementado el rebote de la pelota en Vaus podríamos llegar a tener un juego que nunca termina. Por ejemplo, porque queda sólo un ladrillo en pantalla, pero dado el ángulo de la bola, podríamos estar golpeándola todo el rato sin conseguir romper dicho ladrillo. Para evitar esta situación (y de paso simular mejor el juego original), haremos que cuando la pelota sea golpeada con Vaus en movimiento (tecla izquierda o derecha pulsada), se le aplique un efecto al rebote. Realmente esto ocurre también en la vida real: no es lo mismo golpear una pelota con la raqueta en movimiento que con la raqueta en reposo.

Este es un extracto del código que aplica el efecto indicado:

```
if (inputStates.right)
            ball.angle = ball.angle * (ball.angle < 0 ? 0.5 : 1.5);
else if (inputStates.left)
            ball.angle = ball.angle * (ball.angle > 0 ? 0.5 : 1.5);
```

STOP

Debes decidir dónde incluir el código anterior y comprobar que la bola rebota con efecto cuando la golpeas mientras mueves la raqueta.

15. Power-Ups - Bonus

Una vez roto, un ladrillo puede dejar caer un regalo o bonus con una cierta probabilidad (e.g. 1/100). Ese regalo tendrá diferente color dependiendo del tipo de bonus liberado[14]. Por ejemplo:

- Rojo: puedes disparar y romper cualquier tipo de ladrillo (salvo los irrompibles). Los ladrillos plateados requerirán más disparos. Si implementas enemigos, también puedes dispararles (**L**áser).
- Azul celeste (cyan): la bola actual se divide en tres, es decir, habrá 3 bolas en juego a la vez. En el juego original, éste era el único bonus que, mientras lo usabas, provocaba que no cayeran otros (**D**isruption).
- Azul oscuro: alarga la anchura de Vaus (**E**nlarge)

Puedes implementar cualquiera de estos bonus usando los gráficos que encontrarás en la hoja de sprites. El tipo de bonus actual puede reflejarse o bien en el marcador o bien en el color o gráfico de la raqueta. Ten en cuenta que en la hoja de gráficos, los sprites de los bonus no están situados en horizontal. Tenemos varias opciones para solucionar esto:

- modificar la librería Sprites.js para que acepte gráficos que ocupan varias filas,
- editar la hoja de gráficos para situar los sprites de los bonus en horizontal,
- usar otra hoja de gráficos auxiliar,
- usar sólo alguno de los sprites disponibles (los que formen una única fila).

En este último punto de la segunda parte de la práctica optaremos por la opción más sencilla: usar sólo los sprites de bonus que ocupen una fila. Además, implementaremos sólo el esqueleto de funcionamiento de los bonus para aprender a mostrarlos en pantalla y hacerlos caer. El resto de funcionalidades queda como trabajo opcional.

Lo primero que haremos será crear una clase para construir y gestionar (mover y pintar) Bonus. Es muy parecida a la función para construir y gestionar bolas[15]:

```
 function Bonus () {
    this.type= 'C'; // para este ejemplo, sólo un tipo y marcado a fuego
(hard-coded)
    this.x= 50; // para este ejemplo, el bonus saldrá de esta posición.
```

[14] Puedes ver una descripción completa de los distintos tipos de Bonus o Power-Ups aquí.
[15] Aunque no se solicita, piensa cómo podrías abstraer una `superclase' o investiga el concepto de \textit{mixins}. Puede ser una buena refactorización que contaría como trabajo opcional. El usuario no lo verá, pero el programador que toque tu código ¡lo agradecerá!

```
En el juego
      this.y= 50; //  debería salir al azar desde la posición del
ladrillo roto más reciente
    this.width= 16;  // ancho y alto
    this.height= 8;
    this.speed= 80; // velocidad, puedes trastear para ajustarla a lo
que más te guste
    this.sprite= new Sprite('img/sprites.png', [224,0], [16,8], 0.5,
[0,1,2,3]);
      // las coordenadas del bonus verde, su anchura y su altura.
      // Queremos una animación lenta (0.5) y usar sólo 4 frames, del 0
al 3.
      };

      Bonus.prototype = {
      draw : function(ctx) {
                  ctx.save();
                ctx.translate(this.x,this.y);
            this.sprite.render(ctx); // pintar el bonus en su posición
x,y
            ctx.restore();
      },
      move : function() {
            this.sprite.update(delta); // apuntar al nuevo frame de la
animación
            this.y += calcDistanceToMove(delta, this.speed); // mover el
bonus hacia abajo
      }
  };
```

Sólo nos queda instanciar la clase, añadir el bonus a un array de posibles bonuses y pintarlo en su nueva posición en cada vuelta del mainloop. Es decir,

```
  ...
   var bonuses = []; // declarar e inicializar a vacío un array de bonus
al comienzo de GF
  ...
   bonuses.push(new Bonus()); // incluir esta línea a la función
startNewGame()
  ...
   updateBonus(); // incluir esta llamada dentro del mainLoop
```

La función updateBonus() se limita a recorrer todos los posibles bonus del array (puede haber más de un bonus en juego) y para cada uno de ellos, moverlo (move) y pintarlo (draw). Si implementas correctamente los puntos anteriores, deberías ver algo como lo

siguiente (el bonus verde caerá hasta el fondo de la pantalla, con una animación en la que parece que gira... eso sí, una animación un tanto hosca, dado que sólo hemos usado una fila de sprites para ello, cuando el bonus real tiene dos):

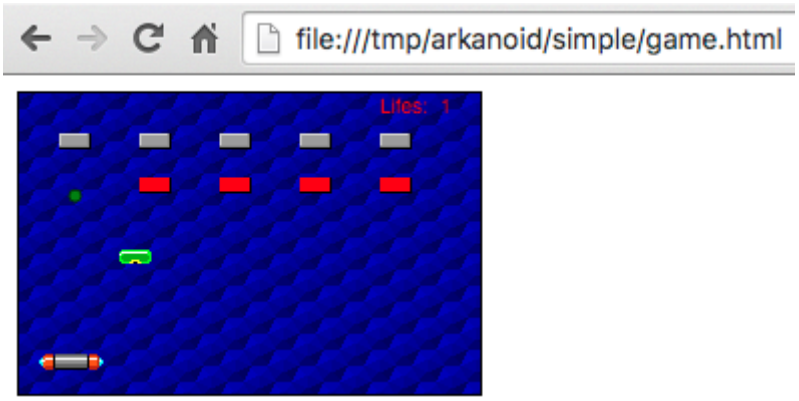


Figura 4: El bonus verde cae mientras parece que gira sobre sí mismo.

Tercera parte

16. 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. Sal a celebrarlo :)

Cuando programas tu primer juego complejo, siempre te surgen nuevas ideas y opciones de mejora. De hecho, el Arkanoid 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 excedas! Te recomiendo comentarlo conmigo (email, tutorías, ...) antes de implementar alguna de estas funciones extra.

16.1. Ladrillos especiales

En la versión actual todos los ladrillos son iguales salvo en el color. Pero en la versión original del juego, no sucede así. En esa versión, algunos ladrillos son irrompibles (los de color plateado) y otros tienen que ser golpeados varias veces para ser rotos (los de color oro). Además, la puntuación por romper los distintos tipos de ladrillos depende del color del ladrillo. Los ladrillos de color plateado y oro tienen una animación al ser golpeados.

16.2. Niveles

Añade nuevos niveles de juego. Cuando el jugador haya roto todos los ladrillos automáticamente pasará de nivel. Cada nivel tendrá distintos muros de ladrillos[16]. Además, cada nivel tiene un fondo distinto al anterior (puedes ver los fondos disponibles en la imagen de sprites). Si implementas esta opción, deberías mostrar por pantalla el nivel actual en todo momento.

16.3. Puntuación

Modifica tu juego de tal forma que se muestre una puntuación mientras el usuario está jugando. Puedes elegir tu propia estrategia de puntuación dependiendo del tipo de ladrillo que hayas roto. También puedes mostrar el High Score (máxima puntuación obtenida por el

[16] Ver http://strategywiki.org/wiki/Arkanoid/Walkthrough

mejor jugador hasta el momento) y las vidas que le quedan al jugador actual (en forma de pequeños gráficos Vaus en la parte inferior izquierda de la pantalla en lugar de un simple número como hasta ahora).

16.4. Bonus - Power-Ups

Hemos implementado un esqueleto básico del funcionamiento de los bonus. Ahora queda lo más interesante: detectar el momento en el que la raqueta intercepta el bonus y aplicar el efecto adecuado dependiendo del tipo de *power-up* (rojo=láser, verde=multibolas, negro=vida-extra, ...). Al recoger el bonus se puede aplicar un efecto de sonido (usando alguno de los sprites de sonido de sounds.mp3). El gráfico de Vaus también podría variar en función del bonus.

16.5. Pausar el juego

Modifica tu código para que el jugador pueda pulsar la tecla"p" o "P" cuando quiera pausar el juego. Para volver a jugar deberá pulsar la misma tecla.

16.6. 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 la música original asociada a esta pantalla.

16.7. Pruebas unitarias con QUnit desde consola

Las pruebas unitarias se han realizado usando QUnit desde JSFiddle. Es posible usar QUnit directamente desde consola, mediante la ayuda de node.js, Bower o Grunt. En este ejercicio opcional el objetivo sería crear un único archivo con todas las pruebas unitarias QUnit y ejecutarlo desde consola para comprobar rápidamente si alguna de ellas falla.

16.8. 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 Internet Explorer. Otra opción sería implementar una versión para móviles, donde usando los eventos de Device Orientation y Device Motion[17] el jugador pudiera mover a Vaus inclinando el móvil a un lado u otro (y disparar pulsando en pantalla[18]).

16.9. Enemigos

Sí, el Arkanoid original también tenía enemigos. Al igual que con los bonus, encontrarás los gráficos de animación de los enemigos en la hoja de sprites.

---

[17] https://web.dev/learn/

[18] Mediante gestión de eventos touch