

Práctica: PAC-MAN

20 de marzo de 2022



Índice

1. Introducción y reglas básicas	1
1.1. Interacción del usuario	1
1.2. Fantasmas	1
1.3. Panel de puntuación	2
1.4. Niveles	2
1.5. Diseño de la Práctica	2
2. Esqueleto básico de un juego	3
3. Midiendo los FPS	4
3.1. ¿A cuántos FPS se refresca la pantalla?	4
3.2. Mover Pacman a izquierda y derecha	5
4. Eventos de teclado	6
5. Gestión del mapa de nivel	6
5.1. Dibujando el mapa de nivel	7
6. Moviendo a Pacman por el mundo virtual	8
7. El poder de las píldoras	9
7.1. Píldoras de poder	10
8. Puertas de teletransportación	10
8.1. Fantasmas	11
8.2. Choque con fantasmas	13
9. Cazando fantasmas	13
9.1. Reloj	14
9.2. Recogiendo píldoras de poder	14
9.3. Pintando fantasmas vulnerables	14
9.4. Vuelta a casa	14
10. Congelando el tiempo	15
11. Control de las vidas	16
11.1. Game Over	17
12. Música y efectos sonoros	18
13. Bonus points (1 punto)	18
13.1. Dos jugadores	18
13.2. Juego en red	18
14. Evaluación	19

1. Introducción y reglas básicas

Pac-Man es un juego de arcade (máquina recreativa) desarrollado por Namco y publicado por primera vez en Japón en mayo de 1980. Fue creado por el diseñador de videojuegos japonés Toru Iwatani. Junto al Tetris (1984), es uno de los videojuegos más conocidos a nivel mundial. Pac-Man es considerado uno de los clásicos y un icono de la cultura pop de los 80. Nada más publicarse, se convirtió en un fenómeno social que se tradujo en grandes ventas de material relacionado con el videojuego (*merchandising*) y de otros videojuegos inspirados en él. Pac-Man forma parte de la colección dedicada a videojuegos del Museo de Arte Moderno (MoMA) de Nueva York y del Smithsonian, en Washington.

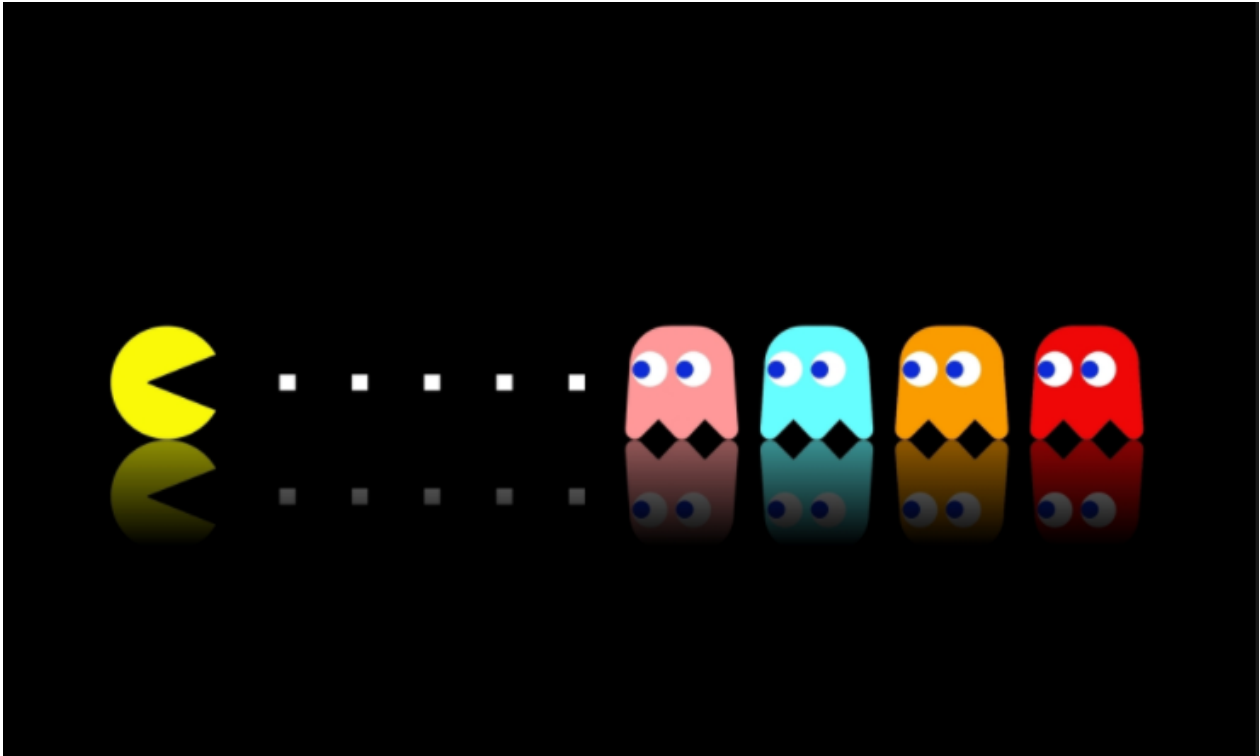


Figura 1: Pacman, las píldoras y los 4 fantasmas del juego

1.1. Interacción del usuario

El usuario puede usar las flechas del teclado (\leftarrow , \rightarrow), (\uparrow), (\downarrow) para mover a pacman.

El objetivo de cada nivel es comer todas las píldoras que se encuentren en el mismo, sin ser cazado por los fantasmas.

En cada nivel encontrarás además 4 píldoras de poder, algo más grandes y de otro color que las píldoras normales. Si Pacman come una de ellas, se convertirá en un Pacman con el poder suficiente como para cazar a los fantasmas. De hecho, los fantasmas se volverán azules cuando Pacman tenga este poder. Al comerlo, el fantasma muere y su alma regresa a la casa de los fantasmas, donde resucitará.

1.2. Fantasmas

Los fantasmas son criaturas que intentarán cazar a Pacman y lo matarán si lo cogen.



Figura 2: Los 4 fantasmas

En el juego original, cada fantasma tiene su propia estrategia para perseguir y coger a Pacman.

INKY (Azul). Se intentará situar siempre en frente de la boca de Pacman. **BLINKY (Rojo)**. Es el más agresivo de los 4 fantasmas. Comenzará a perseguir a Pacman desde el principio, y se dirigirá directamente hacia él. Su casa es la esquina superior derecha. **PINKY (Rosa)**. Al igual que Inky, se intentará situar siempre en la boca de Pacman. **CLYDE (Naranja)**. Se desconoce su estrategia real.

1.3. Panel de puntuación

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 Pacman. 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 pequeños Pacman.

Una píldora equivale a 10 puntos. Una píldora de poder equivale a 50 puntos. Comerse un fantasma equivale a 100 puntos.

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 laberinto de cada nivel). Para ello, el primer paso será construir dicho laberinto, que puede ser diferente para cada uno de los niveles (como en el juego original).

1.5. Diseño de la Práctica

Para ayudarte con el diseño de la práctica, dispondrás de un fichero de ayuda (plantilla en JavaScript) 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 'Tu código aquí'. Debes borrar dicho comentario e implementar en su lugar cada método en cuestión. Para que la versión del Pacman 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:

```
1 var addStarToTheBody = function() {  
2     document.body.innerHTML += "★";  
3     // calls again itself AFTER 200ms  
4     setTimeout(addStarToTheBody, 200);  
5 };  
6  
7 // calls the function AFTER 200ms  
8 setTimeout(addStarToTheBody, 200);
```

Esta forma de animación no es óptima. Si estamos tratando con un juego, como es el caso, *setTimeout* seguirá ejecutándose¹ 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*.

```
1 window.onload = function init() {  
2     requestAnimationFrame(mainloop);  
3 };  
4  
5 function mainloop(timestamp) {  
6     document.body.innerHTML += "★";  
7  
8     // call back itself every 60th of second  
9     requestAnimationFrame(mainloop);  
10 }
```

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 podría suponer un problema.

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)². Ese objeto dispone de un método público *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).

¹A mucha menor velocidad

²Este objeto GF está basado en el ofrecido por el curso 'HTML5 Part 2: Advanced Techniques for Designing HTML5 Apps', muy recomendado para I@s alumn@s de DAWE, <https://www.edx.org/course/html5-part-2-advanced-techniques-w3cx-html5-2x-1>

```

1  var GF = function() {
2    var mainLoop = function(time) {
3      // Función Main, llamada en cada frame
4      requestAnimationFrame(mainLoop);
5    };
6    var start = function() {
7      requestAnimationFrame(mainLoop);
8    };
9    // Nuestro GameFramework sólo muestra una función pública al exterior (el mé
    todo start)
10   return {
11     start: start
12   };
13 };

```

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

```

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

```



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 verde, fondo verde, en una posición aleatoria del canvas. Comprueba que tu aplicación pasa las pruebas unitarias del ejercicio (test1).

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*.

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)³. ¿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).

³Realmente 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>

```

1  // variables para contar frames/s, usadas por measureFPS
2  var frameCount = 0;
3  var lastTime;
4  var fpsContainer;
5  var fps;
6  var measureFPS = function(newTime){
7      // la primera ejecución tiene una condición especial
8      if(lastTime === undefined) {
9          lastTime = newTime;
10         return;
11     }
12     // calcular el delta entre el frame actual y el anterior
13     var diffTime = newTime - lastTime;
14     if (diffTime >= 1000) {
15         fps = frameCount;
16         frameCount = 0;
17         lastTime = newTime;
18     }
19     // mostrar los FPS en una capa del documento
20     // que hemos construido en la función start()
21     fpsContainer.innerHTML = 'FPS: ' + fps;
22     frameCount++;
23 };

```

Llamaremos a la función *measureFPS* desde mainloop:

```

1  var mainLoop = function(time){
2
3      measureFPS(time);
4
5      // llamar a mainloop cada 1/60 segundos
6      requestAnimationFrame(mainLoop);
7  };

```



Modifica el método *drawPacman()* para pintar a Pacman en pantalla en la posición x,y que llega como parámetro. Pacman tendrá por el momento un color amarillo, con un perfil negro. La boca estará abierta con un ángulo inicial de 0.25 radianes y final de 0.75 radianes (radio, 15). El centro de Pacman debe estar situado en (10+radio,100+radio). Comprueba que tu aplicación pasa las pruebas unitarias del ejercicio (test2). Verifica que en tu ordenador de mesa consigues unos 60fps.





3.2. Mover Pacman a izquierda y derecha

Nuestro siguiente objetivo será mover Pacman 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 nuestro personaje “rebote” al lado contrario (por el momento, más adelante modificaremos este comportamiento).



Implementa el método *move* que mueve Pacman de izquierda a derecha según se ha indicado. Pacman debe empezar a moverse en la posición marcada por sus atributos (*posX*, *posY*). Comprueba que tu aplicación pasa las pruebas unitarias del ejercicio (*test3*). ¡OJO! Para pasar las pruebas unitarias, debes desactivar el *mainLoop* (comenta la llamada *requestAnimationFrame* dentro de ese loop)

4. Eventos de teclado

Para poder mover a Pacman con las flechas 'Izquierda' , 'Derecha' , 'Arriba'  y 'Abajo' , debemos añadir código para gestionar los eventos de teclado, es decir, conocer cuándo el usuario ha pulsado una tecla. Para ello definiremos un objeto llamado *inputStates* con los atributos *left*, *right*, *up*, *down* y *space* (este último nos permitirá poner el juego en pausa). Estos atributos tomarán valores booleanos en función de si la tecla correspondiente está pulsada o no. Ahora el método *move* de Pacman -que se ejecuta en el *mainloop*- incluirá una comprobación del estado de *inputStates* para saber si debe moverse el personaje a izquierda, derecha, arriba o abajo (la tecla de pausa la dejaremos para más adelante).

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



Atención

Debes pasar las pruebas unitarias del *test4*. Ten en cuenta que los atributos *posX* y *posY* del pacman ahora se llaman *x* e *y*. El objeto pacman ahora se llama *player*. Las pruebas unitarias simulan que Pacman empieza en la posición superior izquierda y que el usuario ha pulsado la tecla derecha (Pacman se moverá hasta chocar contra la pared derecha, momento en el cual, se detendrá). La siguiente prueba simula que el usuario ha pulsado la tecla abajo (hasta tocar el fondo). La siguiente simula una pulsación de la tecla izquierda (hasta tocar el borde izquierdo). La última simula la pulsación de la tecla arriba (hasta tocar el borde superior). Recuerda actualizar también el método *move* para que Pacman no vaya de izquierda a derecha automáticamente, sino que actualice la posición *x* o *y* dependiendo de si está dentro de los límites del canvas. Nota: Cuando el usuario pulse espacio basta con que muestres por consola un mensaje.

5. Gestión del mapa de nivel

Vamos a cargar el mapa de nivel a partir de un fichero de texto. Este fichero (*res/levels/1.txt*) representa el nivel 1 del juego del Pacman. Si te fijas, las primeras líneas guardan datos generales del nivel:

```
# lvlwidth 21
# lvlheight 25
```


La altura y la anchura vienen representadas en unidades baldosa (*tiles*). Una baldosa representa un cuadrado del mapa (podemos considerar el mapa como un puzzle de piezas rectangulares, donde cada pieza es una baldosa).

Las siguientes líneas forman el mapa del nivel en sí:

```
# startleveldata
0 0 0 0 0 0 0 0 0 0 113 21 113 0 0 0 0 0 0 0 0
0 107 100 100 100 100 100 100 100 100 106 0 105 100 100 100 100 100 100 100 100 100
0 101 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 101 0
...
0 101 2 120 2 107 112 2 111 100 100 100 112 2 111 108 2 120 2 101 0
0 101 2 2 2 101 2 2 0 0 4 0 0 2 2 101 2 2 2 101 0
0 105 133 112 2 110 2 111 100 100 133 100 100 112 2 110 2 111 133 106 0
...
101 3 120 2 2 2 2 2 2 2 2 2 2 2 2 2 2 120 3 101
101 2 2 2 107 100 100 100 100 108 0 107 100 100 100 100 108 2 2 2 101
105 100 100 100 106 0 0 0 0 110 21 110 0 0 0 0 105 100 100 100 106
# endleveldata
```

Cada número representa un componente del juego. Por ejemplo, el número 4 representa a Pacman, el 0 una baldosa vacía, el 2 una píldora, el 3 una píldora de poder, los números entre 100 y 199 (incluidos) representan distintos trozos de pared y entre 10 y 13 (incluidos) a los fantasmas.

Queremos cargar el nivel actual en un array llamado *map*⁴ una instancia de la clase *Level*.



Implementa la clase *Level* y sus métodos *loadLevel*, *setMapTile* y *getMapTile*, de tal forma que el siguiente trozo de código:

```
1 var thisLevel = new Level(canvas.getContext("2d"));
2 thisLevel.loadLevel( thisGame.getLevelNum() );
3 thisLevel.printMap();
```

cargue en el array *map* de la clase *level* el nivel 1.txt. La clase *thisGame*, un objeto literal que viene ya implementado en la plantilla, permite guardar datos generales sobre el juego: número de nivel en el que nos encontramos, tamaño de la baldosas, etc.

El método *printMap()* es opcional, no se prueba en los tests unitarios, pero es un buen método auxiliar para comprobar que la carga del mapa es correcta. Debería imprimir por consola el contenido del atributo *map*, formateándolo en *lvlheight* filas y *lvlwidth* columnas.

Tu código debe pasar las pruebas unitarias del test5. Es recomendable usar alguna librería que te ayude a cargar ficheros de texto (jQuery, por ejemplo).

5.1. Dibujando el mapa de nivel

En esta sección dibujaremos en pantalla el mapa que acabamos de cargar. Por ahora no nos vamos a preocupar de que Pacman encaje en la rejilla del mapa (es decir, por ahora Pacman se moverá igual que si no hubiera un mapa, con total libertad). Definiremos un método *drawMap* en la clase *Level* que, haciendo uso del array *map* pinte en pantalla un nivel como el de la figura 3.

⁴Unidimensional, de *lvlWidth***lvlheight* casillas.

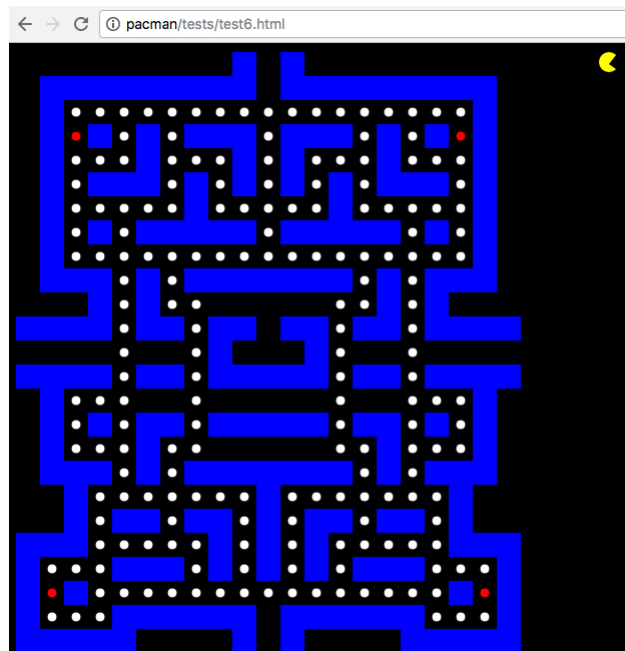


Figura 3: Pintando el nivel 1

En este nivel, las paredes son azules, el fondo negro, las píldoras blancas y las píldoras de poder rojas. Fíjate también que, al igual que en el juego original, hay 4 puertas especiales (en el centro de las esquinas superior, inferior, derecha, izquierda).

Cuando Pacman entre por la puerta derecha saldrá automáticamente por la izquierda (y viceversa). Lo mismo ocurre con las puertas superior e inferior. Como se ha indicado, este control de movimientos se llevará a cabo más adelante.

El método *drawMap* debe pintar todas las baldosas del nivel (nuestra pantalla siempre será como máximo de *thisGame.screenTileSize[0]* x *thisGame.screenTileSize[1]* baldosas). Para saber qué tipo de baldosa debemos pintar (vacía o puerta especial - fondo negro -, pared - fondo azul -, píldora - círculo blanco - o píldora de poder - círculo rojo-, basta con hacer uso del método *getMapTile* (que programamos en el test anterior) y comprobar el tipo de baldosa. En función de ello, debes pintar un rectángulo (del color adecuado) o un pequeño círculo (blanco o rojo). Una vez que termines, asegúrate de que el código pasa los test unitarios del test6. Nota: fíjate que el radio de Pacman ha cambiado (ya no es 15, sino 10). También la altura y anchura del canvas, así como el fondo de pantalla (negro).

6. Moviendo a Pacman por el mundo virtual

Pacman debe moverse por el mundo virtual que hemos pintado en pantalla en el ejercicio anterior. Para ello, debemos:

- Colocar a Pacman en el centro de la pantalla. Al **cargar el mapa**, ten en cuenta que el número de baldosa con ID=4 representa a Pacman. Deberás inicializar los atributos *homeX*, *homeY* de nuestro personaje dependiendo de la fila y columna donde encuentres ese valor (recuerda también las dimensiones de una baldosa: *TILE_WIDTH* x *TILE_HEIGHT*)
- Deberás crear un método *reset* que inicialice la posición x,y de Pacman a *homeX*, *homeY* al comienzo del juego (realmente, siempre que el jugador pierda una vida, como veremos más adelante)

- Crear un método que detecte si un posible movimiento de Pacman hará que éste se choque contra alguna pared (método *checkIfHitWall*). A su vez, este último método necesitará el método auxiliar *isWall* que, tomando como parámetros (fila, col) determina si hay una pared en esa posición del mapa.
- Modificar el método *move* de Pacman para que sólo se mueva allá donde no haya paredes



Implementa los métodos que se han indicado y comprueba que tu aplicación pasa las pruebas unitarias del ejercicio (test7). Ojo, la velocidad (speed) de Pacman es ahora de 3 (era 5).



Atención

Aunque pases el test7 comprobarás que es bastante difícil mover a Pacman cuando éste quiere pasar entre dos paredes: la pulsación de teclado debe ser exactamente en el momento en el que Pacman enfila el pasillo por el que quiere pasar. Esto es sumamente complejo. Es más, si Pacman está moviéndose de izquierda a derecha y el jugador intenta pulsar la tecla arriba para entrar por un pasillo vertical, si no acierta, Pacman se parará, lo que hace que el juego vaya “a trompicones”. Para evitar ambos efectos no deseables (e incrementar enormemente la jugabilidad), el juego original recurría a una ingeniosa técnica: basta con hacer que Pacman no se pare cuando el jugador intenta moverse a un pasillo y falla (ha pulsado la tecla demasiado pronto o demasiado tarde). En estos casos, Pacman debe continuar moviéndose en la dirección que llevaba, sin pararse. De hecho, dado que una pulsación de tecla genera múltiples asignaciones a *inputStates* (la comprobación se hace 60 veces por segundo, y una pulsación de tecla suele durar bastante más de 1/60 segundos), cuando el jugador intenta mover a Pacman y se adelanta demasiado en la pulsación, dado que Pacman sigue su camino y la tecla permanece pulsada durante unos milisegundos, Pacman entrará por la abertura deseada inmediatamente (el jugador creerá que ha acertado de pleno, aunque no haya sido así).

En resumen, en el método *checkInputs*, antes de actualizar las velocidades *x* e *y* de Pacman cuando el usuario pulse una tecla, asegúrate de que esa actualización es posible (esa actualización no provoca que el jugador choque contra una pared). Si no es posible, no actualices las velocidades (seguirán siendo las que eran y Pacman seguirá su camino).

7. El poder de las píldoras

Al pasar por encima de las píldoras blancas, Pacman debe recogerlas. Esto hará que ocurran varias cosas:

- La píldora desaparece del mapa. Para ello, basta con actualizar el atributo *map* convenientemente, sustituyendo el valor de la píldora (*pellet*) por el valor 0 en la fila, columna adecuada (que calcularás a partir de la posición *x,y* actual de Pacman).

- Decrementar el contador de píldoras en juego (atributo *pellets* de la clase *Level*).
- La puntuación se incrementa (trataremos la puntuación más adelante)
- Si es la última píldora en pantalla, pasaremos de nivel (esto también se gestionará más adelante).



Implementa el método *checkIfHitSomething* de la clase *Level*. Este método toma como parámetros la posición x,y de Pacman y la fila y columna más cercana y comprueba si ha chocado contra una píldora. En tal caso, se actualiza el estado tal y como se ha indicado. Por ahora, si el jugador termina de comer todas las píldoras, imprimir en consola el texto "Next level!". El método *checkIfHitSomething* será invocado desde el método *move* de la clase *Pacman*, tras haber actualizado su posición x,y. Comprueba que tu aplicación pasa las pruebas unitarias del ejercicio (test8).

7.1. Píldoras de poder

Primero, vamos a darle un pequeño toque especial al dibujado de las píldoras de poder. Hasta ahora simplemente se pintaban en pantalla como un círculo rojo. Vamos a añadir un pequeño efecto de parpadeo, es decir, que parezca que se encienden y se apagan. Para ello, añadiremos un atributo *powerPelletBlinkTimer* que funcionará como un contador. Cada vez que pintemos el mapa, este contador se incrementará (módulo 60). Sólo pintaremos la píldora de poder cuando este contador sea menor de 30. Esto creará el efecto buscado.



Atención

No hay pruebas unitarias para este ejercicio, pero no pases al siguiente hasta no terminar este.

8. Puertas de teletransportación

En Pacman hay cuatro puertas especiales, a las que denominaremos puertas de teletransportación, en los laterales de la pantalla. En este ejercicio, debes modificar el método *checkIfHitSomething* de la clase *Level* para determinar si Pacman está atravesando una de dichas puertas. Usa la estructura de datos *tileID* del método *checkIfHitSomething* y el método auxiliar *getMapTile* para determinar si Pacman se encuentra sobre una baldosa de tipo puerta horizontal ('door-h') o puerta vertical ('door-v'). En tal caso, mueve a Pacman a la otra esquina del mapa (pero no lo dejes justo ahí, porque en tal caso la lógica del juego determinará que Pacman está otra vez en una puerta de teletransportación y volverá a moverlo, repitiendo este proceso de forma indefinida). Es decir, muévelo a la otra esquina del mapa y avanza su posición x o y el tamaño de una baldosa (para sacarlo de la zona de teletransportación y evitar el efecto indeseado que se acaba de definir).



Modifica el método *checkIfHitSomething* de la clase *Level* para dar soporte de teletransportación, siguiendo el algoritmo que se ha indicado. Comprueba que tu aplicación pasa las pruebas unitarias del ejercicio (test9).

8.1. Fantasmas

Llegó la hora de implementar el código necesario para pintar en pantalla y mover a Clyde, Blinky, Pinky e Inky (!).

Dentro de la clase GF definiremos algunas variables relacionadas con la gestión de fantasmas en el juego.

```
1
2     var numGhosts = 4;
3     var ghostcolor = {};
4     ghostcolor[0] = "rgba(255, 0, 0, 255)";
5     ghostcolor[1] = "rgba(255, 128, 255, 255)";
6     ghostcolor[2] = "rgba(128, 255, 255, 255)";
7     ghostcolor[3] = "rgba(255, 128, 0, 255)";
8
9     // los siguientes dos colores los usaremos más adelante, al implementar la
       captura de píldoras de poder
10    ghostcolor[4] = "rgba(50, 50, 255, 255)"; // blue, vulnerable ghost
11    ghostcolor[5] = "rgba(255, 255, 255, 255)"; // white, flashing ghost
12
13    // create ghost objects
14    var ghosts = {};
```

Definiremos también una clase Ghost, con atributos para guardar la posición, velocidad y estado (normal, vulnerable, muerto) de cada fantasma. La clase Ghost tendrá dos métodos principales, draw() y move().

La firma del constructor Ghost es la siguiente:

```
Ghost = function(id, ctx)
```

Donde id representa número entero, entre 0 y numGhosts-1, que nos permitirá asignar un identificador a cada fantasma. ctx representa el contexto del canvas donde pintar los fantasmas.

```
draw = function()
```

Este método debe pintar el fantasma en el contexto del canvas. Recuerda que el color de cada fantasma está definido en el array *ghostcolor*. Por lo tanto, necesitarás la siguiente línea en algún momento:

```
// ctx es el contexto del canvas, le llega al constructor Ghost como parámetro
this.ctx.fillStyle = ghostcolor[this.id];
```

Para pintar el fantasma, te recomiendo usar dos curvas cuadráticas. Puedes realizar pruebas en esta web: [Interactive quadraticcurveto](#). ¿Por qué dos curvas y no una? Si lo consigues hacer sólo con una, ¡perfecto!, pero yo no he sido capaz :) Para pintar el fantasma he creado dos curvas cuadráticas solapadas (lo que hace que al fantasma le salgan dos pequeños cuernos redondeados) Para pintar los ojos, puedes valerte del método arc() del canvas (con dos arcos blancos en forma de círculo).

```
move = function()
```

Este método debe intentar mover el fantasma en una dirección, hasta encontrar una bifurcación. En caso de bifurcación, elegir al azar la dirección a tomar (entre las posibles). El algoritmo que seguimos en este método es el siguiente:

```
1
2     si el fantasma está exactamente situado sobre una baldosa (no está en
      medio de dos baldosas)
3     calcular fila de baldosa, columna de baldosa sobre la que está situado
4     posible movimientos: [ [0,-1], [1,0], [0,1], [-1,0] ];
5     soluciones= []
6     por cada posible movimiento
7         si el movimiento es posible en el nivel actual desde la pos actual (
          no chocamos contra una pared)
8             a~adir movimiento a soluciones
9     si hemos chocado contra una pared o si hay una bifurcación
10        elegir una dirección (un movimiento) al azar del array de soluciones
11        actualizar velocidades (velX y velY) convenientemente
12
13 si no
14     incrementar x, incrementar y con velX, velY (uno de los dos
      incrementos será 0, claro)
```

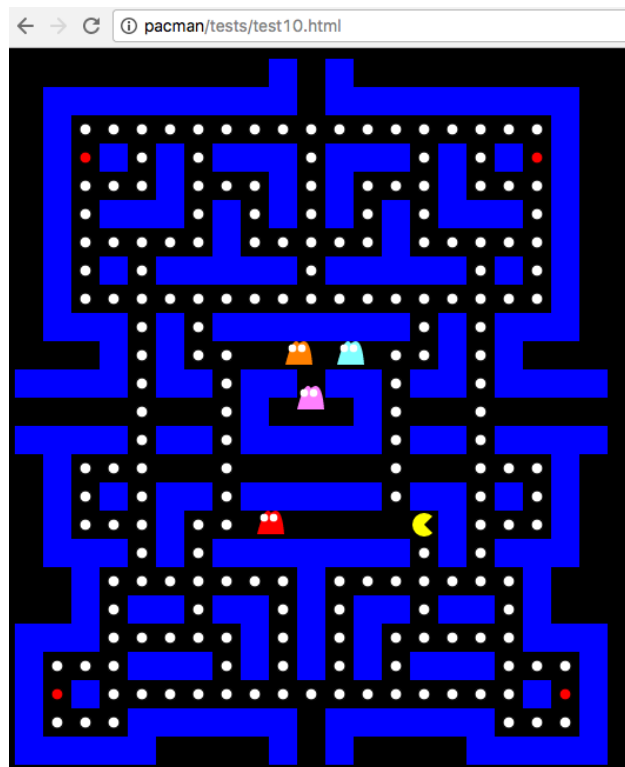


Figura 4: Pintando fantasmas

Justo debajo del código que instancia a Pacman, añadiremos un bucle para instanciar a nuestros cuatro fantasmas:

```
1  for (var i=0; i< numGhosts; i++){
2      ghosts[i] = new Ghost(i, canvas.getContext("2d"));
3  }
```



Crea la clase `Ghost` con los métodos *draw* y *move* tal y como se ha indicado. Debes cargar también la posición inicial de los fantasmas en este nivel (házlo dentro del método *loadLevel*). Añade al método *mainLoop* las líneas necesarias para mover y pintar los 4 fantasmas en cada frame. Finalmente, inicializa el valor de los atributos *x*, *y*, *velX*, *velY* de los fantasmas de forma conveniente en el método *reset()*. Asegúrate de pasar las pruebas unitarias del *test10*. Si todo va bien, los fantasmas deberían aparecer en pantalla, moviéndose al azar por el mapa, tal y como se ve en la figura 4.

8.2. Choque con fantasmas

Modificaremos el método *move* de Pacman para controlar el momento en el que nuestro personaje choque contra uno de los 4 fantasmas. Lo haremos tras comprobar que no hay colisión con otros elementos, llamando al método *checkIfHit* de la clase *Level*. Este método tiene la siguiente firma:

```
this.checkIfHit = function(playerX, playerY, x, y, holgura)
```

Donde (playerX,playerY) representa la posición de Pacman y (x,y) la del fantasma que estamos comprobando. Holgura es un valor real que determina cuánto debe "meterse" un personaje dentro del otro para considerar que han chocado, tanto en horizontal como en vertical. Por ejemplo, si la holgura es de 3 y sólo se "tocan" 1 pixel ($|playerX - x| > holgura$ y $|playerY - y| > holgura$), no deberíamos considerar que han chocado. Ajusta la holgura como más te convenga, pero para pasar el test unitario deja la holgura en un valor de `TILE_WIDTH/2`. Por el momento, cuando Pacman choque contra un fantasma, nos limitaremos a escribir por consola que han chocado.



Implementa el código indicado y asegúrate de pasar las pruebas unitarias del *test11*.

9. Cazando fantasmas

Hemos dibujado en pantalla las píldoras de poder, pero no hemos implementado su función. En esta sección solucionaremos este aspecto. Cuando Pacman recoge una píldora de poder los fantasmas deben pasar a un estado vulnerable, es decir, Pacman podrá cazarlos. Este estado vulnerable durará unos 6 segundos. Cuando los fantasmas son vulnerables su color es azul durante los primeros 5 segundos. El último segundo su color parpadeará entre azul y blanco, lo que permite a Pacman actuar en consecuencia (por ejemplo, ir a cazarlo si el fantasma está cerca o alejarse si no lo está). Cuando Pacman caza un fantasma, su alma regresará a la casa de los fantasmas, donde resucitará. Representaremos el alma de un fantasma por medio de sus ojos ⁵, es decir, veremos que al ser cazado, son sus ojos (o gafas) los que regresan a casa.

Empezaremos programando el código necesario para capturar las píldoras de nivel y cambiar el color de los fantasmas según lo indicado.

⁵hay gente que dice que realmente son las gafas del fantasma - *spectacles* en inglés -

9.1. Reloj

Lo primero que haremos será declarar un reloj (contador), de nombre *ghostTimer* en el objeto literal *thisGame*. Este reloj será el que nos permita medir los 6 segundos que marcan la vulnerabilidad de los fantasmas.

Añade también un método *updateTimers* a la clase *GF*. Este método decrementa (si es posible) en una unidad el contador *ghostTimer*. Si el contador alcanza el valor 0, el estado de los fantasmas pasará de *Ghost.VULNERBLE* a *Ghost.NORMAL*. El método *updateTimers* debe ser invocado en cada vuelta del *mainLoop*.

9.2. Recogiendo píldoras de poder

Debes modificar también el método *checkIfHitSomething* de la clase *Level*. Simplemente añade una nueva rama condicional para que si Pacman ‘choca’ contra una píldora de poder ésta desaparezca del mapa (al igual que hacemos con las píldoras normales) y el estado de los 4 fantasmas pase de *Ghost.NORMAL* a *Ghost.VULNERABLE*. Además, inicializaremos el valor de *ghostTimer* a 360 (6 segundos, a razón de 1 segundo cada 60 fps).

9.3. Pintando fantasmas vulnerables

Modifica el método *draw* de la clase *Ghost*. En concreto, gestiona el color del fantasma de la siguiente forma:

1. Si el estado es *Ghost.NORMAL*, pinta el fantasma como lo hacías hasta ahora (no cambia nada)
2. Si el estado es *Ghost.VULNERABLE*, comprueba el valor del *ghostTimer*. Si es mayor que 100, pinta el fantasma de azul (recuerda que este color está definido en la variable *ghostcolor[4]*). Si es menor de 100, debes hacer que parpadea entre azul y blanco (recuerda que el blanco está definido en *ghostcolor[5]*).



I Implementa el código indicado y asegúrate de pasar las pruebas unitarias del test12.

9.4. Vuelta a casa

Cuando cazamos un fantasma vulnerable (*Ghost.VULNERABLE*), haremos que sus ojos (o gafas) vuelvan a su posición de origen. Durante este tiempo el fantasma estará en modo *Ghost.SPECTACLES*.

Modifica el método *move* de *Pacman* para que cuando se detecta una colisión contra un fantasma, en caso de que éste esté en el estado *Ghost.VULNERABLE* su estado pase a ser *Ghost.SPECTACLES* y su velocidad X e Y se modifiquen acordemente para que comience a regresar en línea recta a su posición de partida.

En el método *draw* de *Ghost*, añade una condición para que si el estado del fantasma es *Ghost.SPECTACLES* no se pinte el cuerpo del mismo (sólo los ojos), tal y como se ve en la figura 5.

En el método *move* de *Ghost*, añade una condición para que si el estado del fantasma es *Ghost.SPECTACLES* el movimiento de los ojos del fantasma sea en línea recta ⁶ hasta su posi-

⁶o lo más recto posible, que no ande pululando por el mapa

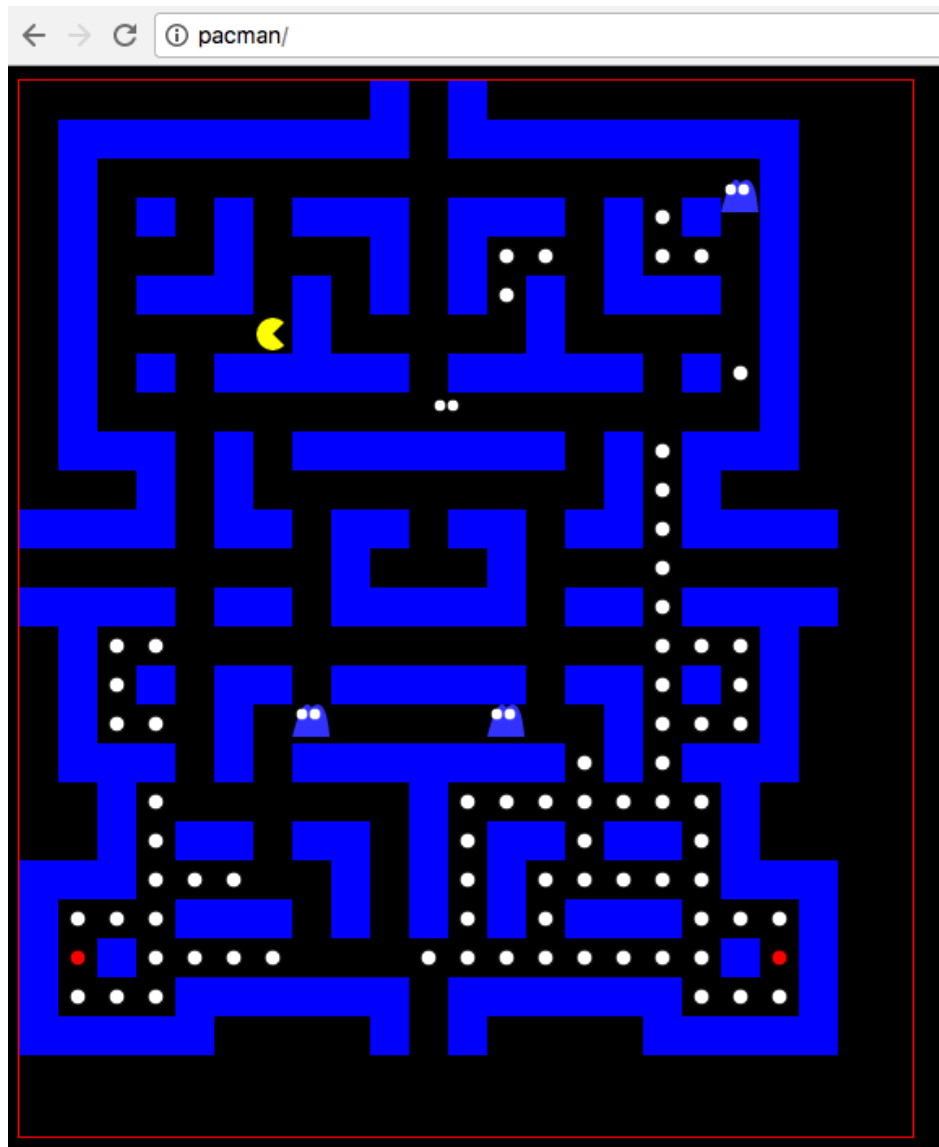


Figura 5: Pacman ha cazado a un fantasma que vuelve a casa

ción original (*homeX*, *homeY*). Cuando alcancen esa posición, el estado del fantasma debe volver a ser *Ghost.NORMAL*.



| Implementa el código indicado y asegúrate de pasar las pruebas unitarias del test13.

10. Congelando el tiempo

Hasta el momento, cuando Pacman choca contra un fantasma lo único que hemos hecho ha sido detectar el golpe y escribir en consola un mensaje. En esta sección mejoraremos esa implementación simulando el comportamiento del juego original. En concreto, cuando Pacman choque el juego debe congelarse durante 1.5 segundos, para que el jugador sea consciente de lo que ha ocurrido. Pasado ese tiempo, decrementaremos en una unidad el número de vidas (asumimos que partimos

con tres vidas) y resetaremos la posición y velocidad de los fantasmas y de Pacman a su valor inicial. Esperaremos ahora 0.5 segundos antes de que el juego vuelva a comenzar.

Para realizar todas las tareas indicadas, comenzaremos por añadir un atributo *mode* al juego (objeto literal *thisGame*).

Este atributo puede tomar los siguientes valores:

1. **NORMAL** Situación normal del juego, Pacman y los fantasmas se mueven libremente, como hasta ahora.
2. **HITGHOST** Pacman ha golpeado a un fantasma. Congelamos todos los movimientos en pantalla para que el jugador sea consciente del golpe. Resetamos posiciones.
3. **WAIT_TO_START** Tras resetear las posiciones de los fantasmas y de Pacman, esperamos 0.5 segundos antes de volver a empezar.

Para los momentos de espera indicados (1.5 segundos al chocar y 0.5 segundos antes de volver a empezar tras el reset) usaremos un contador (reloj) : *thisGame.modeTimer*. La gestión de los distintos modos de juego la realizaremos dentro del *mainLoop*.

En cualquier caso, en el método *updateTimers* incrementaremos en una unidad el atributo *thisGame.modeTimer*.

Recuerda que debes modificar el método *move* de *Pacman* para que al chocar contra un fantasma, establezca el modo de juego a **HIT_GHOST**.

Cada vez que realices un cambio de modo de juego, debes llamar al método *setMode* del objeto *thisGame*. Este se encargará no sólo de establecer el nuevo modo de juego (al que le llegue como parámetro), sino que reiniciará el contador *modeTimer* a 0.



| Implementa el código indicado y asegúrate de pasar las pruebas unitarias del test14.

11. Control de las vidas

Debemos gestionar las vidas del jugador. Inicialmente partiremos con 3 vidas y cada vez que un fantasma capture a Pacman perderemos una. Si llegamos a cero vidas se acabará el juego (Game Over).

Crea una nueva variable llamada *lives* dentro de *thisGame* e inicialízala a 3. A continuación, modifica el método *mainLoop* para que antes de resetear el estado del juego porque Pacman ha sido capturado, se le reste una vida.

Crea también dos variables más en *thisGame*: una llamada *points*, que deberá llevar la puntuación del juego. Otra llamada *highscore*, que llevará la puntuación máxima alcanzada por algún jugador (por ahora sólo gestionaremos *points* y dejaremos el control del *highscore* para la segunda parte de la práctica).

Crea un método *displayScore* que pinte en pantalla la puntuación, las vidas y la puntuación máxima (por el momento siempre 0). Debería quedarte algo similar a lo siguiente:

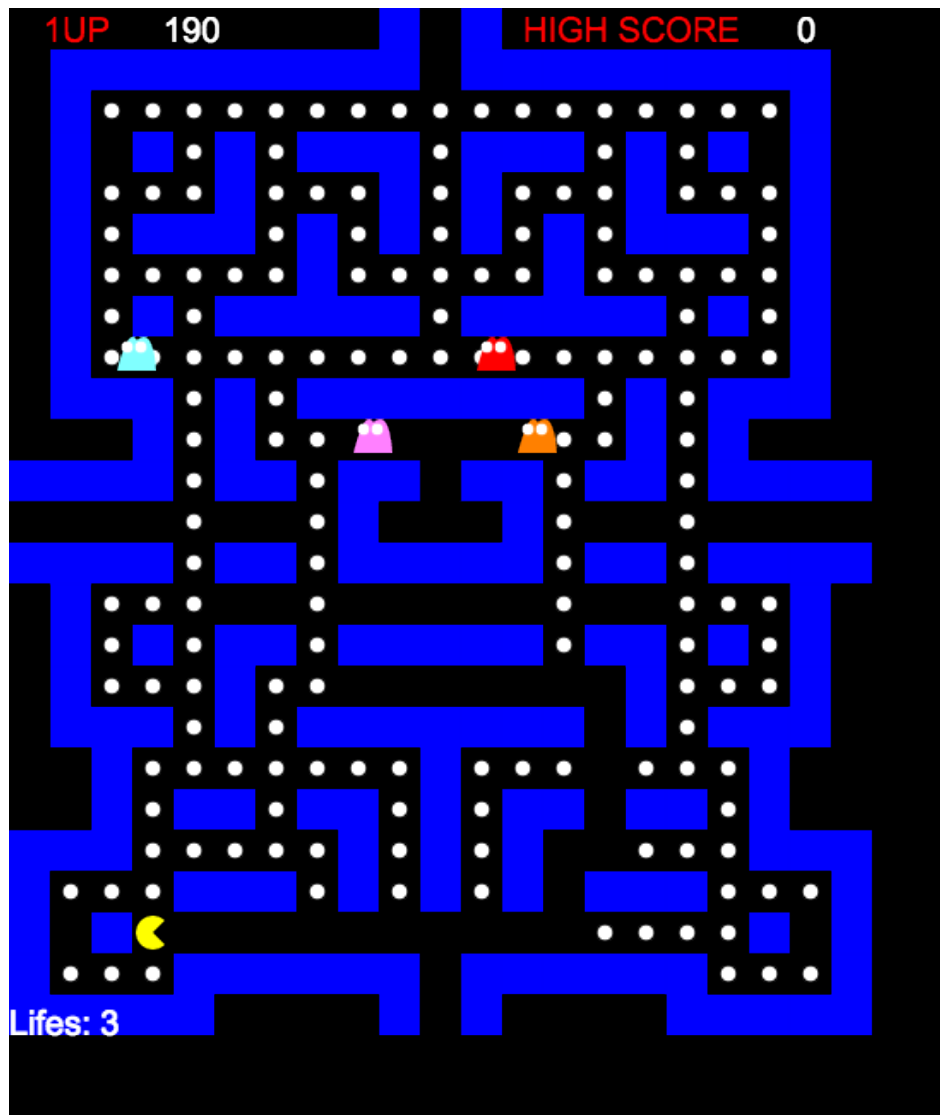


Figura 6: Debes mostrar la puntuación, vidas y highscore. Esta es una opción, pero elige tu propio diseño :)



Atención

No hay pruebas unitarias para este ejercicio ni para el siguiente, pero ¡no entregues esta parte de la práctica sin haber implementado el control de vidas y puntuación básico!

Importante: Cada vez que necesites actualizar la puntuación, hazlo llamando al método `addToScore(puntos_a_sumar de thisGame`. NO actualices la puntuación directamente. Esto nos permitirá programar fácilmente en el futuro la opción de dar vidas extra por haber alcanzado cierta cantidad de puntos.

11.1. 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 Pacman y de los fantasmas. Crea un nuevo estado en *thisGame* llamada `GAME_OVER`. Dentro del *mainLoop* comprueba este nuevo estado para saber si hay que parar el juego y pintar en pantalla el famoso mensaje de `GAME OVER` :)



¡ENHORABUENA!

Has terminado de programar tu propio juego Pacman :-) 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.

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?

12. Música y efectos sonoros

Añade una música de fondo al juego. Junto a ello te propongo añadir un efecto sonoro para cada una de estas situaciones:

- Pacman se come una píldora.
- Pacman se choca contra una pared.
- Pacman se come a un fantasma (y este muere).
- Pacman pierde una vida.
- El juego se termina (game over).

13. Bonus points (1 punto)

¿Quieres sacar un diez? Para ello te propongo a continuación algunas funcionalidades que te ayudarán a lograr que el juego sea más entretenido. No hay ayuda en el código de la plantilla para implementar estas sugerencias pero merece la pena intentarlo, ¿no crees?.

13.1. Dos jugadores

Se trata de que dos jugadores puedan jugar a la vez con un único teclado y pantalla. Un poco incómodo si el teclado es pequeño, pero puede servir como base para la siguiente propuesta.

13.2. Juego en red

Realmente esta sería la forma idónea para dar soporte a dos jugadores. Hemos estudiado cómo trabajar con WebSockets, comunicar datos por JSON, gestionar eventos... Dispones de todo lo necesario, ¿te animas a afrontar este reto? :)

14. Evaluación

Los tests son una ayuda para autoevaluar si se está desarrollando el juego de acuerdo a lo que se propone pero no es necesario adjuntar los ficheros que demuestran que se ha superado cada uno de ellos. La evaluación se centrará en el análisis de la jugabilidad, es decir, si el juego cumple con las funcionalidades propuestas en el enunciado.

La creatividad también se va a tener en cuenta: en caso de desarrollar otras funcionalidades que no se proponen en la práctica esto también se valorará.

Para la entrega de esta tarea se pide subir a eGela un documento en PDF, .odt o .doc con al menos los siguientes apartados:

- Nombre y Apellidos.
- URL pública al juego desarrollado (en tu droplet de DigitalOcean).
- URL al repositorio GitHub (compártelo con el usuario @AinhoY).
- Cualquier aclaración que se desee exponer sobre la implementación realizada y/o descripción de algún aspecto nuevo desarrollado si lo hubiera. Esto no es obligatorio.