Videojuegos con HTML5 (Canvas). Animaciones CSS3

1. Objetivos

En este quinto tema se pretenden conseguir los siguientes objetivos:

- Conocer el nuevo elemento <canvas> de HTML5.
- Utilizar el elemento <canvas> para crear objetos gráficos.
- Crear videojuegos simples utilizando el elemento <canvas>.
- Crear animaciones sin necesidad de programar utilizando las animaciones de CSS3.
- Controlar la reproducción de las animaciones CSS3 mediante JavaScript. Encadenar animaciones CSS3.

2. Introducción

En el tema anterior vimos cómo crear interfaces de usuario utilizando los nuevos elementos para formularios introducidos con HTML5. También vimos cómo controlar la validación de estos formularios mediante la nueva API Forms. En este quinto tema, veremos un elemento no menos importante, el nuevo elemento <canvas> de HTML5, creado para cubrir las carencias que teníamos anteriormente para el dibujo de objetos gráficos y el desarrollo de videojuegos sin necesidad de utilizar plug-ins externos.

Nos centraremos, sobre todo, en la creación de videojuegos sencillos (dejando un poco de lado el manejo de gráficos), para lo cual, no tendremos más remedio que utilizar JavaScript, pero trabajaremos sobre un ejemplo sencillo y no nos meteremos en detalles demasiado complejos.

Asimismo, en temas anteriores vimos cómo crear transiciones (pequeñas animaciones) sobre elementos de nuestra página. Veremos en este tema, cómo encadenar estas transiciones utilizando para ello las animaciones CSS3. Además, veremos cómo podemos controlar la reproducción de estas animaciones y encadenar una tras otra, utilizando para ello los nuevos eventos para animaciones incorporados al lenguaje JavaScript.

1. Videojuegos con HTML5

Al comienzo del curso hablamos sobre cómo HTML5 está reemplazando complementos o plug-ins anteriores, como Flash o Java applets, por ejemplo.

Había dos cosas importantes a considerar para independizar a la web de tecnologías desarrolladas por terceros: procesamiento de video y aplicaciones gráficas. El elemento <video> y la API para medios cubren el primer aspecto muy bien, pero nos faltaría cubrir el tema de los gráficos y los videojuegos. La API Canvas se hace cargo del aspecto gráfico y lo hace de una forma extremadamente efectiva. Canvas nos permite dibujar, presentar gráficos en pantalla, animar y procesar imágenes y texto, y trabaja junto con el resto de la especificación para crear aplicaciones completas e incluso videojuegos en 2 y 3 dimensiones para la web.

Esta API ofrece una de las más poderosas características de HTML5. Permite a desarrolladores trabajar con un medio visual e interactivo para proveer capacidades de aplicaciones de escritorio para la web.

El elemento <canvas>

Este elemento genera un espacio rectangular vacío en la página web (lienzo), en el cual serán mostrados los resultados de ejecutar los métodos provistos por la API. Al crearlo, obtendremos únicamente un espacio en blanco, como un elemento vacío, pero con un propósito totalmente diferente.

```
<canvas id="lienzo" width="500" height="300">
Su navegador no soporta el elemento canvas
</canvas>
```

Como se puede observar en el ejemplo, sólo es necesario especificar unos pocos atributos para este elemento. Los atributos width (ancho) y height

(alto) declaran el tamaño del lienzo en píxeles. Estos atributos son necesarios debido a que todo lo que sea dibujado sobre el elemento tendrá esos valores como referencia. Siempre indicaremos un atributo id, para poder acceder fácilmente al elemento desde el código JavaScript.

Básicamente, lo que hace el elemento <canvas> es crear una caja vacía en la pantalla. Después, a través de JavaScript, utilizaremos los nuevos métodos y propiedades introducidos por la API para sacarle el máximo partido a esta superficie.

Por razones de compatibilidad, en caso de que el navegador no soporte este elemento, el contenido entre las etiquetas <canvas> será mostrado por pantalla.

Acceder al elemento <canvas> desde JavaScript

Para acceder al elemento <canvas> desde JavaScript, en primer lugar, accederemos al objeto DOM del elemento, normalmente, a través del método getElementById, y posteriormente, utilizaremos el método getContext para obtener el contexto de dibujo, sobre el cual, podremos dibujar.

```
var lienzo=null, canvas=null;
function iniciar() {
    canvas=document.getElementById('lienzo');
    lienzo=canvas.getContext('2d');
}
window.addEventListener("load", iniciar, false);
```

En el ejemplo, guardamos una referencia al elemento <canvas> en la variable elemento y el contexto de dibujo lo creamos utilizando

getContext('2d'). Con esto obtenemos un contexto para dibujar en 2 dimensiones. Las últimas versiones de los navegadores, ya soportan el contexto en 3 dimensiones, pero por motivos obvios de tiempo, no lo veremos en este curso.El contexto de dibujo del lienzo será una tabla de píxeles listados en filas y columnas de arriba a abajo y de izquierda a derecha, con su origen (el píxel 0,0) ubicado en la esquina superior izquierda del lienzo.

Dibujando en el lienzo

Una vez que tenemos acceso al contexto del <canvas> podemos comenzar a crear y manipular gráficos. La API Canvas dispone de una extensa lista de herramientas para este propósito, desde la creación de simples formas y métodos de dibujo, hasta texto, sombras o transformaciones complejas. No vamos a estudiar con detenimiento las posibilidades gráficas que nos ofrece este API, pero sí que veremos las nociones básicas necesarias para poder desarrollar un pequeño videojuego.

Dibujando rectángulos

Normalmente, el desarrollador deberá preparar la figura a ser dibujada en el contexto, pero existen algunos métodos que nos permiten dibujar directamente en el lienzo, sin preparación previa. Estos métodos son específicos para formas rectangulares y son los únicos que generan una forma primitiva (para obtener otras formas tendremos que combinar otras técnicas de dibujo y trazados complejos). Para el propósito que nos ocupa, nos bastará con ver el método **fillRect(x, y, ancho, alto)** que dibuja un rectángulo relleno.

La esquina superior izquierda estará ubicada en la posición especificada por los parámetros x e y. Los parámetros ancho y alto declaran el tamaño.

```
var lienzo=null, canvas=null;
function iniciar() {
        canvas=document.getElementById('lienzo');
        lienzo=canvas.getContext('2d');
        lienzo.fillRect(110,110,100,100);
}
window.addEventListener("load", iniciar, false);
```

En el ejemplo, el contexto fue asignado a la variable global lienzo, y ahora usamos esta variable para referenciar el contexto en cada método de dibujo.

El método fillRect(110,110, 100,100) del ejemplo, dibuja un rectángulo relleno, comenzando desde la posición 110,110 del lienzo y con un ancho y un alto de 100 píxeles, es decir, se tratará de un cuadrado.

Colores

Hasta el momento, hemos usado el color por defecto (negro), pero podemos especificar el color que queremos aplicar utilizando la propiedad **fillStyle**. Esta propiedad declara el color para el interior de la figura (el relleno).Podemos modificar el ejemplo anterior de la siguiente forma para cambiar el color del cuadrado:

```
var lienzo=null, canvas=null;
function iniciar() {
        canvas=document.getElementById('lienzo');
        lienzo=canvas.getContext('2d');

        lienzo.fillStyle="#000099";
        lienzo.fillRect(110,110,100,100);
}
window.addEventListener("load", iniciar, false);
```

En el ejemplo, los colores fueron declarados usando números hexadecimales. También podemos usar funciones como rgb(), o incluso, especificar transparencia para la figura aprovechando la función rgba(). Estos métodos deben ser escritos entre comillas (por ejemplo, **fillStyle="rgba(255,165,0,1)**").

Cuando especificamos un nuevo color este se convertirá en el color por defecto para el resto de los dibujos, a menos que volvamos a cambiarlo más adelante.

Gradientes

Al igual que en CSS3, los gradientes en la API Canvas pueden ser lineales o radiales, y pueden incluir puntos de terminación para combinar colores:

- **createLinearGradient(x1, y1, x2, y2):** este método crea un objeto que luego será usado para aplicar un gradiente lineal al lienzo.
- **createRadialGradient(x1, y1, r1, x2, y2, r2):** este método crea un objeto que luego será usado para aplicar un gradiente circular o radial al

lienzo usando dos círculos. Los valores representan la posición del centro de cada círculo y sus radios.

- **addColorStop(posición, color):** este método especifica los colores que usará el gradiente. El atributo posición es un valor entre 0.0 y 1.0 que determina dónde la degradación comenzará para ese color en particular.

```
var lienzo=null, canvas=null;
function iniciar() {
        canvas=document.getElementById('lienzo');
        lienzo=canvas.getContext('2d');

        var gradiente=lienzo.createLinearGradient(0,0,10,100);
        gradiente.addColorStop(0.5, '#0000FF');
        gradiente.addColorStop(1, '#000000');
        lienzo.fillStyle=gradiente;

        lienzo.fillRect(10,10,100,100);
}
window.addEventListener("load", iniciar, false);
```

Como vemos, creamos el objeto gradiente desde la posición (0,0) a la (10,100), otorgando una leve inclinación hacia la izquierda. Los colores los declaramos con el método addColorStop() y el gradiente logrado se aplica a la propiedad fillStyle, como si se tratara de un color regular.

Es importante tener en cuenta que las posiciones del gradiente corresponden al lienzo, no a las figuras que queremos afectar. Por lo tanto, si movemos el rectángulo dibujado a una nueva posición, el gradiente cambiará.

Dibujar textos

Escribir texto en el lienzo es tan simple como definir unas pocas propiedades y llamar al método apropiado. Para configurar como se insertará el texto disponemos de tres propiedades:

- **Font**: Esta propiedad tiene una sintaxis similar a la propiedad font de CSS, y acepta los mismos valores.
- **textAlign**: Esta propiedad alinea el texto. Existen varios valores posibles: start (comienzo), end (final), left (izquierda), right (derecha) y center (centro).
- **textBaseline**: Esta propiedad es para alineamiento vertical. Establece diferentes posiciones para el texto. Los posibles valores son: **top, hanging, middle, alphabetic, ideographic y bottom**.

Para dibujar un texto con las letras rellenas utilizaremos el método **fillText(texto, x, y)**. Del mismo modo que el método fillRect para el trazado, este método dibujará el texto especificado en la posición x, y como una figura rellena. Puede también incluir un cuarto valor para declarar el tamaño máximo. Si el texto es más extenso que este último valor, será encogido para caber dentro del espacio establecido.

Mi mensaje

```
var lienzo=null, canvas=null;
function iniciar() {
        canvas=document.getElementById('lienzo');
        lienzo=canvas.getContext('2d');

        lienzo.font="bold 24px verdana, sans-serif";
        lienzo.textAlign="start";
        lienzo.fillText("Mi mensaje", 100,100);
}
window.addEventListener("load", iniciar, false);
```

Como podemos ver en el ejemplo, la propiedad font puede tomar varios valores a la vez, usando exactamente la misma sintaxis que CSS. En este caso la propiedad textAling hace que el texto sea dibujado desde la posición 100,100 (si el valor de esta propiedad fuera end, por ejemplo, el texto terminaría en la posición 100,100). Finalmente, el método fillText dibuja un texto sólido en el lienzo.Existen otros métodos para trabajar con textos en el API Canvas, pero con este tendremos suficiente para nuestro propósito.

Mostrar imágenes

En el API Canvas disponemos del método **drawImage** para trabajar con imágenes. Este método nos permitirá dibujar una imagen en el lienzo. Sin embargo, puede recibir un número de valores que producen diferentes resultados. Estudiemos estas posibilidades:

- **drawImage(imagen, x, y):** Esta sintaxis es para dibujar una imagen en el lienzo en la posición declarada por x e y. El primer valor es una

referencia a la imagen que será dibujada.

- **drawImage(imagen, x, y, ancho, alto)**: Esta sintaxis nos permite escalar la imagen antes de dibujarla en el lienzo, cambiando su tamaño con los valores de los parámetros ancho y alto.
- **drawImage(imagen, x1, y1, ancho1, alto1, x2, y2, ancho2, alto2)**: Esta es la sintaxis más compleja. Hay dos valores para cada parámetro. El propósito es cortar partes de la imagen y luego dibujarlas en el lienzo con un tamaño y una posición específica. Los valores x1 e y1 declaran la esquina superior izquierda de la parte de la imagen que será cortada. Los valores ancho1 y alto1 indican el tamaño de esta pieza. El resto de los valores (x2, y2, ancho2 y alto2) declaran el lugar donde la pieza será dibujada en el lienzo y su nuevo tamaño, el cual puede ser igual o diferente al original. Este método viene muy bien cuando tenemos sprites de imágenes, pero nosotros no los vamos a utilizar.

En todos los casos, el primer atributo puede ser una referencia a una imagen en el mismo documento generada por métodos como getElementById, o creando un nuevo objeto imagen usando métodos regulares de JavaScript. No es posible usar una URL o cargar un archivo desde una fuente externa directamente con este método.

```
var lienzo=null, canvas=null;
function iniciar() {
    canvas=document.getElementById('lienzo');
    lienzo=canvas.getContext('2d');
```

```
        var imagen=new Image();
        imagen.src="imgs/fruit.png";
        imagen.addEventListener("load", function(){
            lienzo.drawImage(imagen,20,20)
        }, false);
}
window.addEventListener("load", iniciar, false);
```

El ejemplo anterior lo único que hace es cargar la imagen y dibujarla en el lienzo. Debido a que el lienzo sólo puede dibujar imágenes que ya están completamente cargadas, necesitamos controlar esta situación escuchando al evento load de la imagen. Agregamos una escucha para este evento y declaramos una función anónima para responder al mismo. El método drawImage dentro de esta función dibujará la imagen cuando esté completamente cargada.

Organizando el código del videojuego

Para comenzar con nuestros videojuegos, vamos a organizar un poco nuestro código, de forma que la operación de dibujo la sacaremos a una función que llamaremos paint y que recibirá el lienzo como parámetro.

```
var lienzo=null, canvas=null;
function iniciar() {
        canvas=document.getElementById('lienzo');
        lienzo=canvas.getContext('2d');
        paint(lienzo);
}
```

```
function paint(lienzo) {
        lienzo.fillStyle='#0f0';
        lienzo.fillRect(50,50,100,60);
}
window.addEventListener("load", iniciar, false);
```

Animar el canvas

En la primera parte del tema, hemos visto cómo dibujar en nuestro lienzo. Eso está bien, pero cuando desarrollamos un juego, se trata de interactuar con los objetos, no solo de dibujarlos. Por tanto, necesitamos darle movimiento a nuestros objetos gráficos.

Para mostrar la técnica haremos que nuestro rectángulo se vaya desplazando horizontalmente a lo largo del canvas. Primero, declararemos dos nuevas variables globales (x e y) y las inicializaremos.

```
var x=50,y=50;
```

A continuación, modificaremos nuestra función paint para que limpie la pantalla antes de volver a dibujar en ella (esto lo haremos dibujando un rectángulo del tamaño completo del lienzo). Posteriormente, dibujaremos de nuevo nuestro rectángulo, pero con las coordenadas actualizadas (haremos el rectángulo más pequeño para que el efecto sea más claro). Para mejorar un poco el aspecto de nuestro videojuego, dibujaremos el rectángulo relleno con un gradiente.

```
function paint(lienzo) {
        var gradiente=lienzo.createLinearGradient(0,0,0,canvas.height);
        gradiente.addColorStop(0.5, '#0000FF');
        gradiente.addColorStop(1, '#000000');
        lienzo.fillStyle=gradiente;
        lienzo.fillRect(0,0,canvas.width,canvas.height);
        lienzo.fillStyle='#0f0';
        lienzo.fillRect(x,y,10,10);
}
```

Como se puede ver, estamos dibujando el rectángulo de fondo de color negro. En este momento, nuestra función iniciar llama sólo una vez a la función paint, por lo tanto, se ejecutará una sola vez, es decir, todo se quedará estático. Si queremos que se comporte como una animación, tenemos que hacer que se llame a la función una y otra vez cada determinado tiempo. Para esto, crearemos una función run(), y sustituiremos la llamada a la función paint que hacemos en iniciar por una llamada a la función run.

```
function run() {
    setTimeout(run, 17);
    act();
    paint(lienzo);
}
```

En la primera línea, llamamos al método setTimeout. Este método, que recibe una función por parámetro, le pedirá al navegador que ejecute dicha función cuando transcurra el número de milisegundos que le indicamos como

segundo parámetro.

Posteriormente, llamamos a una nueva función act(). Hemos visto antes que paint es la función que se encarga de dibujar todo el escenario en nuestro lienzo. La función act la usaremos para realizar todas las acciones que se producen en nuestro juego; en este caso, moveremos nuestro rectángulo sumándole 2 píxeles a nuestra variable x hasta que llegue al borde derecho del canvas momento en el que volverá a valer 0:

```
function act() {
        if(x>canvas.width)
                x=0;
        else
                x+=2;
}
```

Aún cuando las acciones pudieran realizarse en la función paint, es recomendable hacerlo en una función aparte, para una mayor claridad.

Veamos cómo queda nuestro juego:

```
var lienzo=null, canvas=null;
var x=50,y=50;
function iniciar() {
        canvas=document.getElementById('lienzo');
        lienzo=canvas.getContext('2d');

        run();
}
```

```
function run() {
    setTimeout(run, 17);
    act();
    paint(lienzo);
}

function act(){
        if(x>canvas.width)
                x=0;
        else
                x+=2;
}

function paint(lienzo) {
        var gradiente=lienzo.createLinearGradient(0,0,0,canvas.height);
        gradiente.addColorStop(0.5, '#0000FF');
        gradiente.addColorStop(1, '#000000');
        lienzo.fillStyle=gradiente;
        lienzo.fillRect(0,0,canvas.width,canvas.height);
        lienzo.fillStyle='#0f0';
        lienzo.fillRect(x,y,10,10);
}
window.addEventListener("load", iniciar, false);
```

Mejorar el rendimiento de la animación

En el pasado, para crear temporizadores en general, se utilizaba la

función setTimeout. Esta función se ha utilizado durante mucho tiempo para manejar los temporizadores en las páginas web, sin embargo, no fue concebida para realizar animaciones que requieren múltiples llamadas por segundo, consumiendo muchos recursos de nuestro equipo, aun cuando no estamos haciendo uso de la aplicación en cuestión.

Para evitar este problema de rendimiento las compañías desarrolladoras de navegadores web han ideado una mejor solución para esta tarea: la función **requestAnimationFrame**.

Esta función, que recibe otra función como parámetro, le pedirá al navegador que ejecute dicha función la próxima vez que pueda realizar un cuadro de animación (la frecuencia en que se repetirá la ejecución de la función dependerá del rendimiento del equipo, en máquinas actuales la frecuencia de actualización del navegador rondará los 60 cuadros por segundo). Así, mejoraremos la capacidad de manejar animaciones, consumiendo menos recursos, e incluso mandando a dormir el ciclo cuando la aplicación deja de tener el foco.

Para usar **requestAnimationFrame**, la llamaremos en la primera línea de una función, enviando como primer parámetro esa misma función, de forma que la vuelva a llamar después del tiempo de intervalo. Con esto, nuestra función run quedará de la siguiente forma:

```
function run() {
    requestAnimationFrame(run);
    act();
    paint(lienzo);
}
```

requestAnimationFrame(run) equivaldría a llamar a setTimeout(run,17), pero de forma optimizada.

Soporte para navegadores antiguos

requestAnimationFrame es una función relativamente nueva, por lo que los navegadores que no estén actualizados podrían no soportarla[1]. Para poder usar requestAnimationFrame en estos navegadores antiguos, existen muchas soluciones posibles. La más simple y popular, es agregar esta función en nuestro código:

```
window.requestAnimationFrame=(function()
{
        return window.requestAnimationFrame ||
        window.webkitRequestAnimationFrame ||
        window.mozRequestAnimationFrame ||
        function(callback) {
        window.setTimeout(callback,17);
        };
})();
```

Esta función personalizada creará una función requestAnimationFrame con la mejor alternativa posible. Primero intentará utilizar la función estándar.

---

[1] Podéis consultar el soporte actual de esta función por parte de los navegadores en la siguiente url: http://caniuse.com/requestanimationframe.

Si falla, intentará cargar la versión de webkit, que es soportada por versiones antiguas de Safari y de Google Chrome, así como algunos navegadores móviles. Si falla, intentará cargar la versión de Mozilla, que es la que usan las versiones antiguas de Firefox. Finalmente, si no existe soporte para ninguna versión, estándar o experimental, como en el caso de Opera con Presto y las versiones antiguas de Internet Explorer, se creará una llamada clásica a un setTimeout a 17 milisegundos, que es aproximadamente la tasa de actualización de requestAnimationFrame. De esta forma, podremos comenzar a usar requestAnimationFrame en el desarrollo de nuestros videojuegos.

La posible desventaja de usar requestAnimationFrame, es que tiene una tasa de actualización muy pequeña, por lo que nuestros juegos se harán muy lentos en dispositivos de bajo rendimiento como computadoras antiguas y dispositivos móviles. Para prevenir esto, se debe usar un método para regular el tiempo entre distintos dispositivos.

Regular el tiempo entre dispositivos

Para regular el tiempo entre dispositivos se pueden utilizar diversos métodos, cada uno de los cuáles con una serie de ventajas y una serie de inconvenientes.Una posibilidad es hacer de forma asíncrona las funciones paint y act. Esto quiere decir, que cada uno se actualice a su propio ritmo, optimizando así los tiempos para ambas acciones. La desventaja, es que, en algunas ocasiones, necesitaremos que algunos valores interactúen entre las funciones act y paint, lo que podría volverse una tarea un poco más compleja con este método, pero para el videojuego que vamos a crear esto no será un problema.

La forma más sencilla de realizar un método asíncrono es usar un requestAnimationFrame para la función paint y un setTimeout para la función act. Para eso, se crean dos funciones distintas que se llaman a sí mismas continuamente:

```
function run() {
        setTimeout(run,50);
        act();
}
function repaint() {
        requestAnimationFrame(repaint)
        paint(lienzo);
}
```

Lo único que nos quedará por hacer será llamar a ambas funciones (run y repaint) al final de la función iniciar, y el juego funcionará como de costumbre.

Usando el teclado

Nuestro rectángulo ya se mueve por el lienzo, pero para verdaderamente interactuar con él, necesitamos indicarle a dónde queremos que vaya. Para eso, necesitamos primero una variable dónde guardar la tecla presionada:

```
var lastPress=null;
```

También necesitaremos crear un manejador de evento para el teclado que se encargue de almacenar la tecla presionada. El evento que nos interesa en este caso es keydown:

```
document.addEventListener('keydown', function(evt) {
        lastPress=evt.keyCode;
}, false);
```

El manejador lo pondremos al final de nuestro código.A partir de ahora, podremos tomar decisiones en el juego sabiendo la última tecla presionada. Cada tecla tiene un valor numérico, el cual tendremos que comparar para realizar la acción deseada dependiendo la tecla presionada. Como ejemplo, vamos a mostrar por pantalla cuál ha sido la última tecla presionada. Esto lo haremos en nuestra función paint:

```
lienzo.fillText('Last Press: '+lastPress,10,30);
```

En nuestro juego, usaremos las teclas izquierda, arriba, derecha y abajo, cuyos valores numéricos son 37, 38, 39 y 40 respectivamente. Para acceder a ellas más fácilmente, crearemos las siguientes constantes:

```
const KEY_LEFT=37;
const KEY_UP=38;
const KEY_RIGHT=39;
const KEY_DOWN=40;
```

Vayamos ahora al movimiento del rectángulo. Primero necesitaremos definir cuatro constantes para especificar la dirección del rectángulo:

```
const ARRIBA=0;
const DERECHA=1;
const ABAJO=2;
const IZQUIERDA=3;
```

Después, necesitaremos una nueva variable que almacene la dirección de nuestro rectángulo:

```
var dir=DERECHA;
```

El siguiente paso será modificar la dirección que tomará nuestro rectángulo dependiendo de la última tecla presionada (dentro de la función act):

```
if(lastPress==KEY_UP)
	dir=ARRIBA;
if(lastPress==KEY_RIGHT)
	dir=DERECHA;
if(lastPress==KEY_DOWN)
	dir=ABAJO;
if(lastPress==KEY_LEFT)
	dir=IZQUIERDA;
```

A continuación, moveremos nuestro rectángulo dependiendo de la dirección que se haya indicado:

```
if(dir==DERECHA)
	x+=10;
if(dir==IZQUIERDA)
	x-=10;
if(dir==ARRIBA)
	y-=10;
if(dir==ABAJO)
	y+=10;
```

Finalmente, verificaremos si el rectángulo ha salido del canvas, en cuyo caso, haremos que aparezca por el otro lado:

```
if(x>=canvas.width)
        x=0;
if(y>=canvas.height)
        y=0;
if(x<0)
        x=canvas.width-10;
if(y<0)
        y=canvas.height-10;
```

Como vemos, se comprueban todos los bordes del canvas.

Pausar el juego

Otra funcionalidad habitual en los videojuegos es tener la posibilidad de pausarlos cuando se pulsa una determinada tecla. Vamos a implementar esta funcionalidad cuando se pulsa la tecla p. Comenzaremos, por supuesto, creando una variable que indicará si el juego está en pausa:

```
var pause=true;
```

Ahora, encerraremos todo el contenido de nuestra función act en un condicional if(!pause), es decir, si el juego no está en pausa.Después del if, introduciremos el código que se encargará de controlar si el juego está en estado de pausa o no:

```
if (lastPress==KEY_P) {
        pause=!pause;
        lastPress=null;
}
```

Evidentemente, tendremos que indicar cuál es el valor de la constante KEY_P, en este caso 80, por lo tanto, donde definimos las constantes anteriormente introduciremos la siguiente línea:

```
const KEY_P=80;
```

Por último, cuando el juego esté en pausa, dibujaremos en nuestra función paint el texto "PAUSE" centrado:

```
if(pause)  {
lienzo.textAlign='center';
lienzo.fillText('PAUSE',150,75);
lienzo.textAlign='left';
}
```

Ahora, cada vez que presionemos la tecla p, el juego entrará o saldrá de la pausa.

Programación orientada a objetos

En programación se llama "objeto" a un conjunto de propiedades y métodos que definen su comportamiento. Al conjunto de características definidas en base al cual se crea un objeto, se le llama "clase".Los lenguajes de programación permiten crear clases personalizadas, además de aquellas que vienen predefinidas.

JavaScript, a diferencia de otros lenguajes, no tiene clases como tal. Pero se pueden definir funciones que actúan como clases.Para comprender mejor lo aquí explicado, crearemos una clase para objetos de tipo rectángulo.

```
function Rectangle(x,y,width,height,color) {
    this.x=x;
    this.y=y;
    this.width=width;
    this.height=height;
    this.color=color;
}
```

Mediante la función anterior, podemos crear objetos de tipo rectángulo, como se muestra en el ejemplo siguiente:

```
var rect1=new Rectangle(50,50,100,60,"#f00");
```

De esta forma, hemos creado un objeto de tipo rectángulo, con las propiedades inicializadas como sigue: x (50), y (50), ancho (100), alto (60) y color ("#f00").

Si por error se creara un objeto sin especificar todas sus propiedades, las propiedades faltantes obtendrían un valor nulo o indefinido, lo cual podría causar problemas posteriormente en nuestro código. Para prevenir esto, es recomendable que se asigne un valor predefinido en caso que no se especifique alguno de los parámetros:

```
this.x=(x==null)?0:x;
```

En esta línea, cuando x no está definida le asignamos 0. Esta asignación predefinida se repite para los demás valores. En el caso de la altura, cuando su valor sea nulo o indefinido, en lugar de asignarle 0, le asignaremos el mismo valor que el ancho. De esta forma, si envío solo tres valores al rectángulo en lugar de 4, me creará un cuadrado perfecto cuyo ancho y alto será el tercer y último valor asignado:

```
this.height=(height==null)?this.width:height;
```

Es importante asignar this.width y no width directamente, pues de esta forma, se obtiene ya su valor después de comprobarse si era nulo o no. De lo contrario, podríamos asignar un valor nulo por error a la altura, en caso que el ancho enviado haya sido nulo también.

Otra propiedad importante de los objetos, es que tienen métodos propios. Por ejemplo, añadiremos un método a la clase Rectángulo que nos permitirá saber cuándo dicho rectángulo esté en intersección con otro. El

método se llamará intersects y lo definiremos así:

```
this.intersects=function(rect){
        if(rect!=null) {
                return(this.x<rect.x+rect.width &&
                this.x+this.width>rect.x &&
                this.y<rect.y+rect.height &&
                this.y+this.height>rect.y);
        }
        return false;
}
```

Este método se encontrará dentro de la función Rectangle creada anteriormente, y para llamarlo desde fuera de esta, haremos lo siguiente:

```
var colision = rect1.intersects(rect2);
```

Analicemos ahora la función intersects paso a paso. Para empezar, se recibe una variable rect, que será un objeto de tipo rectángulo. En la siguiente línea, comprobamos que el rectángulo no sea nulo, pues en caso de no enviar nada, esto provocaría un error.

Si el rectángulo existe, se ejecuta una comparación de los cuatro puntos de ambos rectángulos, para comprobar si ambos rectángulos se intersectan en algún momento, y se retorna el valor de si esta comparación es cierta o falsa. Otro método útil para nuestro rectángulo, es que este se dibuje automáticamente en nuestro lienzo.

```
this.fill=function(lienzo) {
        if(lienzo!=null) {
                lienzo.fillStyle=this.color;
                lienzo.fillRect(this.x,this.y,this.width,this.height);
                }
}
```

Mediante esta función, sólo debemos indicarle el lienzo donde se dibujará el rectángulo, y este se dibujará de forma automática en donde sus valores indiquen, haciéndolo con una instrucción más sencilla y con menor probabilidad de errar en los valores dados. Al igual que hacíamos con el método intersects el método fill estará dentro de la función Rectangle, y para dibujar un rectángulo desde fuera, sólo tendremos que hacer esto:

```
rect1.fill(lienzo);
```

Nuestra clase Rectangle quedará de la siguiente forma:

```
function Rectangle(x,y,width,height,color) {
	this.x=(x==null)?0:x;
	this.y=(y==null)?0:y;
	this.width=(width==null)?0:width; this.height=(height==null)?
	this.width:height;
	this.color = (color==null)?"#000":color;

	this.intersects=function(rect) {
		if(rect!=null) {
			return(this.x<rect.x+rect.width &&
			this.x+this.width>rect.x &&
			this.y<rect.y+rect.height &&
			this.y+this.height>rect.y);
		}
		return false;
	}

	this.fill=function(lienzo) {
		if(lienzo!=null)  {
			lienzo.fillStyle=this.color;
			lienzo.fillRect(this.x,this.y,this.width,this.height);
		}
	}
}
```

Uso de prototipos

Como hemos visto, JavaScript, a diferencia de otros lenguajes, no tiene clases como tales, pero se pueden usar funciones que trabajen de forma similar a las clases, agregando en su interior las variables y funciones necesarias para que los objetos creados a partir de estas se comporten de la forma deseada.

Sin embargo, el método visto de incluir las funciones en el interior de la clase (que es la forma estándar en que funcionan la mayoría de los lenguajes), no es el más óptimo en JavaScript, pues cada nuevo objeto crea una copia completa de todo su contenido en la memoria RAM, incluyendo las funciones, que en esencia repiten la misma información sin necesidad, y pueden saturar la memoria en caso de trabajar con miles de objetos, como suele ocurrir en los videojuegos.

Para hacer esto correctamente en JavaScript, utilizaremos prototipos en lugar de clases, de esta forma, no se creará una copia de la información en la memoria RAM para cada uno de los objetos creados, permitiendo el uso de múltiples objetos con un impacto menor en la misma.

Usar prototipos puede ser un poco confuso para quienes ya hayan trabajado con otros lenguajes, ya que las funciones deben ser agregadas fuera de la clase, asignándole al prototipo de la misma.

A continuación, veremos cómo quedaría el prototipo de nuestra clase Rectangle:

```
function Rectangle(x,y,width,height,color){
        this.x=(x==null)?0:x;
        this.y=(y==null)?0:y;
        this.width=(width==null)?0:width;
        this.height=(height==null)?this.width:height;
         this.color = (color==null)?"#000":color;
}
```

```
Rectangle.prototype.intersects=function(rect) {
        if(rect!=null) {
                return (this.x<rect.x+rect.width &&
                this.x+this.width>rect.x &&
                this.y<rect.y+rect.height &&
                 this.y+this.height>rect.y);
                }
}

Rectangle.prototype.fill=function(lienzo) {
        if(lienzo!=null) {
                lienzo.fillStyle=this.color;
                lienzo.fillRect(this.x,this.y,this.width,this.height);
        }
}
```

Usar objetos en los videojuegos facilitará muchas de las tareas que tendremos que llevar a cabo. Con esto, concluimos la explicación básica sobre programación orientada a objetos. Si queréis profundizar en el tema de objetos en JavaScript, podéis consultar el siguiente enlace: http://developer.mozilla.org/es/docs/Introducción_a_JavaScript_orientado_a_objetos

Interactuando con otros elementos

Ya hemos visto anteriormente, cómo podemos interactuar nosotros con el juego por medio del teclado, el siguiente punto importante de un juego es que los elementos puedan interactuar entre sí. Por ejemplo, dos elementos que colisionan o un personaje que le dispara a otro. Para saber si dos elementos colisionan, es decir, hay una intersección entre ellos, no sólo nos basta con saber su posición XY, también necesitamos conocer el alto y ancho de los elementos.

Para este propósito, utilizaremos la clase Rectangle creada en el punto anterior. Comenzaremos haciendo unos cambios en nuestro código. En primer lugar, eliminaremos las variables x e y que declaramos al principio, y en su lugar, crearemos una variable que llamaremos **player** de tipo Rectangle:

```
var player=new Rectangle(40,40,10,10,"#0f0");
```

Después, cambiaríamos la forma en que se dibuja el rectángulo: player.fill(lienzo);

Por último, sustituiremos todos los lugares donde usábamos las variables x e y, por el acceso correcto a través del objeto player que hemos creado. Por ejemplo, donde hacíamos **x+=10;** haremos **player.x+=10;**.

Ahora, necesitaremos un nuevo elemento con el cual interactuar. En este caso, vamos a colocar en un punto aleatorio de la pantalla un nuevo rectángulo que hará las veces de comida del rectángulo inicial, de forma que, al pasar con el rectángulo inicial sobre el rectángulo de comida, la puntuación del juego se incremente y la comida se mueva a otro lugar.

Para implementar esta nueva funcionalidad, lo primero que haremos será crear una nueva variable de tipo Rectangle llamada **food**:

```
var food=new Rectangle(80,80,10,10,'#f00');
```

En la función paint, dibujaremos la comida:

```
food.fill(lienzo);
```

Ahora, analizaremos si el objeto player y el objeto food colisionan, en cuyo caso, sumaremos un punto a nuestra puntuación, y cambiaremos la posición de la comida a otro lugar al azar. Para ello, primero tendremos que declarar una variable que nos servirá para almacenar nuestra puntuación:

```
var score=0;
```

También agregaremos una función **random** que nos será muy útil para facilitar el uso de números enteros al azar:

```
function random(max) {
return Math.floor(Math.random()*max);
}
```

Finalmente, en la función act, después de mover a nuestro jugador, comprobaremos si ambos elementos colisionan, y de ser así, sumaremos un punto más y cambiaremos de posición la comida:

```
if(player.intersects(food)) {
        score++;
        food.x=random(canvas.width/10-1)*10;
        food.y=random(canvas.height/10-1)*10;
}
```

La pequeña ecuación de dividir la pantalla entre 10 dentro del random y multiplicarla al final de nuevo, hace que la comida aparezca en un lugar cada 10 píxeles, de esta forma se ajustará "a la rejilla". Por último, dibujaremos nuestra puntuación en pantalla:

```
lienzo.fillText('Score: '+score,10,40);
```

Ahora, cada vez que el rectángulo verde toque al rojo, la puntuación subirá.

Interactuando con varios elementos iguales

Ya hemos visto en el punto anterior cómo hacer que un objeto interactúe con otro. El problema sería si quisiéramos, por ejemplo, querer interactuar con 50 elementos que hagan exactamente lo mismo (imaginemos, por ejemplo un juego del tipo Space Invaders donde todas las naves enemigas son iguales).

Tener que evaluar todos los objetos uno por uno sería demasiado tedioso y complicado. Afortunadamente, hay una forma más sencilla de interactuar con varios elementos de propiedades iguales a través de los arrays.

Vamos a darle una nueva vuelta de tuerca a nuestro videojuego añadiendo elementos de tipo pared que se encontrarán en el escenario. Si nuestro player choca contra un elemento pared el juego terminará. Evidentemente, si queremos que el juego finalice cuando el jugador colisione con una pared, necesitaremos una variable que nos lo indique.

Inicialmente el juego estará en estado **Game Over**.

```
var gameover=true;
```

Para este ejemplo, crearemos un array llamado wall:

```
var wall=[];
```

Este array contendrá todos nuestros elementos de tipo pared. Ahora, agregaremos cuatro elementos a este array de la siguiente forma:

```
wall.push(new Rectangle(100,50,10,10,"#999"));
wall.push(new Rectangle(100,100,10,10,"#999"));
wall.push(new Rectangle(200,50,10,10,"#999"));
wall.push(new Rectangle(200,100,10,10,"#999"));
```

Para dibujar los objetos pared, recorreremos los elementos del array de la siguiente forma:

```
for(var i=0,l=wall.length;i<l;i++) {
    wall[i].fill(lienzo);
}
```

Del mismo modo, comprobaremos si cada elemento pared colisiona con la comida o con el jugador. Esto lo haremos en la función act, después de comprobar la colisión entre la comida y el jugador.

```
for(var i=0;i<wall.length;i++) {
        if(food.intersects(wall[i])) {
                food.x=random(canvas.width/10-1)*10;
                food.y=random(canvas.height/10-1)*10;
        }
        if(player.intersects(wall[i])) {
                gameover=true;
        }
}
```

Primero, comprobamos si la comida colisiona con la pared, en cuyo caso, cambiaremos de lugar la comida, esto evitará que esta quede solapada con la pared. Segundo, comprobamos si el jugador colisiona con la pared, y en tal caso, el juego se detendrá.

Para que el juego se detenga no tenemos que hacer nada en la función act mientras estamos en estado Game Over, por tanto, al condicional que teníamos en esta función (if (!pause)) le añadiremos la comprobación de que no estamos tampoco en Game Over **(if (!pause && !gameover))**.

También queremos que cuando el juego esté en estado Game Over se pinte el texto "Game Over" en la pantalla. Para conseguir este comportamiento, cambiaremos el if(pause) en nuestro paint para ver si el juego está en Game Over o en pausa:

```
if (pause || gameover) {
        lienzo.textAlign='center';
        if(gameover)
                lienzo.fillText('GAME OVER',150,75);
        else
                lienzo.fillText('PAUSE',150,75);
        lienzo.textAlign='left';
}
```

Para terminar, tenemos que implementar alguna manera de reiniciar el juego para que se pueda volver a jugar. Lo que haremos será reiniciar el juego cuando, estando en estado Game Over, se pulse la tecla ENTER. Por lo tanto, tendremos que definir la constante para la tecla correspondiente:

```
const KEY_ENTER=13;
```

A continuación, antes de salir de la función act comprobaremos si el juego ha terminado y se ha pulsado la tecla ENTER, en cuyo caso, lo reiniciaremos:

```
if (gameover && lastPress==KEY_ENTER) {
        reset();
}
```

Nos faltaría la implementación de la función reset:

```
function reset() {
        score=0;
        dir=DERECHA;
        player.x=40;
        player.y=40;
        food.x=random(canvas.width/10-1)*10;
        food.y=random(canvas.height/10-1)*10;
        lastPress=null;
        gameover=false;
}
```

Esta función restaurará el juego a su estado inicial, de forma que, al continuar, se vuelva a comenzar desde el principio. En concreto, pondremos la puntuación a cero, la dirección hacia la derecha, reubicaremos al jugador en su punto inicial y cambiaremos de lugar la comida. Por supuesto, nos aseguraremos de que el juego deje de estar en Game Over.

El juego de la serpiente

Ahora que ya conocemos las bases para hacer un juego, es hora de pulir algunos detalles y darle a nuestro código la forma de uno de los juegos clásicos: El juego de la serpiente.

Lo esencial en el juego de la serpiente, es que el personaje principal va creciendo conforme se alimenta de las manzanas. Para lograr este efecto, utilizaremos un array igual que el de las paredes. Por tanto, sustituiremos de nuestro código la variable player por un array al que llamaremos body.

```
//var player = new Rectangle(40,40,10,10,"#0f0");
var body=[];
```

También tendremos que sustituir cada llamada a player por body[0], que es la cabeza de la serpiente. Así pues, donde decíamos player.x y player.y, ahora diremos body[0].x y body[0].y.

Para que nuestra serpiente tenga siempre la misma longitud al comienzo, debemos resetear el array cada vez que iniciemos el juego. Esto lo haremos en la función reset de esta forma:

```
body.length=0;
body.push(new Rectangle(40,40,10,10,"#0f0"));
body.push(new Rectangle(0,0,10,10,"#0f0"));
body.push(new Rectangle(0,0,10,10,"#0f0"));
```

Es importante que la primera parte del cuerpo (la cabeza) esté en la posición que deseemos que inicie. Las posiciones del resto del cuerpo se calcularán a partir de esta, por tanto, podemos inicializarlas a 0.El siguiente paso será dibujar la serpiente en la función paint, que ya no será sólo la cabeza, sino todo el cuerpo:

```
for(var i=0;i<body.length;i++) {
    body[i].fill(lienzo);
}
```

Para mover el cuerpo de la serpiente, haremos uso de un truco peculiar: lo moveremos de atrás hacia adelante, haciendo así un efecto de oruga, en que la cola va "empujando" el resto del cuerpo:

```
for(var i=body.length-1;i>0;i--) {
        body[i].x=body[i-1].x;
        body[i].y=body[i-1].y;
}
```

Una vez hecho esto, sólo queda mover la cabeza de la serpiente, y nos servirá exactamente el mismo código que teníamos antes para mover el player:

```
if(dir==DERECHA)
      body[0].x+=10;
if(dir==IZQUIERDA)
      body[0].x-=10;
if(dir==ARRIBA)
      body[0].y-=10;
if(dir==ABAJO)
      body[0].y+=10;
```

Para comprobar si el cuerpo choca con la cabeza, lo haremos del mismo modo que comprobábamos si el personaje chocaba con la pared:

```
for(var i=2;i< body.length;i++) {
        if(body[0].intersects(body[i])) {
                gameover=true;
        }
}
```

Llegados a este punto, nos encontramos con un problema, y es que si cuando avanzamos hacia la derecha pulsamos la flecha izquierda, al cambiar el sentido de la marcha, la cabeza choca contra el cuerpo y el juego finaliza. Para solucionar este problema, ignoraremos la pulsación de la tecla inversa a la dirección actual:

```
if(lastPress==KEY_UP && dir != ABAJO)
        dir=ARRIBA;
if(lastPress==KEY_RIGHT && dir!= IZQUIERDA)
        dir=DERECHA;
if(lastPress==KEY_DOWN && dir!=ARRIBA)
        dir=ABAJO;
if(lastPress==KEY_LEFT && dir!=DERECHA)
        dir=IZQUIERDA;
```

El último paso que nos queda es hacer crecer a la serpiente. Esto se hace (lógicamente) cuando la cabeza choca contra la comida:

```
body.push(new Rectangle(0,0,10,10, "#0f0"));
```

Elementos multimedia en los videojuegos

El último punto que veremos sobre videojuegos con HTML5, será cómo incluir medios externos (imágenes y sonido).Actualmente nuestros rectángulos son de 10x10 píxeles, por lo tanto, tendremos que utilizar imágenes del mismo tamaño para que el juego no pierda sentido.

Para representar cada una de las partes de la serpiente utilizaremos la imagen body.png del fichero de recursos, para representar la comida usaremos la imagen fruit.png y para la pared Wall.png. Se guardarán en la carpeta imgs de nuestra web, la ruta que ponemos para cargar la imagen será relativa al documento html donde se encuentra el canvas, no al fichero JavaScript donde se encuentra el código.

Ahora, crearemos nuestras variables de imagen y les asignaremos la ruta a la fuente:

```
var iBody=new Image();
var iFood=new Image();
iBody.src='imgs/body.png';
iFood.src='imgs/fruit.png';
```

Sólo nos queda modificar la función paint, comentando donde dibujamos los rectángulos del cuerpo, la comida y las paredes, y dibujando las imágenes en su lugar:

```
for(var i=0;i<body.length;i++) {
        lienzo.drawImage(iBody, body[i].x, body[i].y);
}
lienzo.drawImage(iFood, food.x, food.y);
for(var i=0,l=wall.length;i<l;i++) {
        lienzo.drawImage(iWall, wall[i].x, wall[i].y);
}
```

Veremos que nuestros gráficos aparecen en la pantalla en lugar de los rectángulos de colores. Podemos hacer algo similar para poner una imagen de fondo a nuestro juego.

Por último, agreguemos algo de sonido. Cada vez que comamos una fruta, reproduciremos un sonido, y al morir, reproduciremos otro.

La declaración será de esta forma:

```
var aComer=new Audio();
var aMorir=new Audio();
aComer.src='sounds/chomp.m4a';
 aMorir.src='sounds/dies.m4a';
```

Para reproducir los sonidos insertaremos las siguientes líneas en las áreas de colisión correspondiente (con la manzana, con el cuerpo y con la pared):

```
aComer.play();
aMorir.play();
```

Como ya vimos en el tema 3, algunos navegadores soportan unos formatos de audio, mientras que otros navegadores soportan otros formatos diferentes. Por tanto, al agregar sonidos a nuestro juego, es posible que en algunos navegadores no se escuche (dependiendo del formato seleccionado).

Hay soluciones avanzadas que permiten descubrir las funciones del navegador del usuario y usar un archivo diferente dependiendo de las mismas. La siguiente función devolverá true si el navegador es capaz de reproducir el formato ogg:

```
function canPlayOgg() {
    var aud=new Audio();
    if(aud.canPlayType('audio/ogg').replace(/no/,''))
        return true;
    else
        return false;
}
```

Este código se implementa de la siguiente forma al asignar la fuente del audio:

```
if(canPlayOgg()) {
        aComer.src="sounds/chomp.ogg";
        aMorir.src='sounds/dies.ogg';
}
else {
        aComer.src="sounds/chomp.m4a";
        aMorir.src='sounds/dies.m4a';
}
```

Controlar la carga de medios externos

Hemos visto anteriormente que no podemos dibujar una imagen en el lienzo si esta no está previamente cargada. En nuestro videojuego la carga será muy rápida, ya que los medios externos son de pequeño tamaño, pero no podemos obviar este punto, ya que en un videojuego algo más elaborado, que trabaje con imágenes o audio de mayor tamaño, tendríamos problemas con esto.

Lo que haremos será pausar la carga del juego hasta que se hayan

cargado todos los medios que se van a utilizar (tanto imágenes como sonidos).

Los medios que necesitemos cargar de forma externa los almacenaremos en un array asociativo que llamaremos medios:

```
var medios = [];
```

En este array iremos introduciendo las imágenes así:

```
medios['iBody'] = new Image();
medios['iBody'].src='imgs/body.png';
medios['iBody'].addEventListener("load", cargaMedio, false);
```

Como vemos, introducimos en el array un objeto de tipo Image, indicamos la propiedad src del mismo (dónde se encuentra la imagen que queremos mostrar) y asociamos un manejador para el evento load que llamará a la función **cargaMedio** que veremos a continuación.

```
var numMediosCargados = 0;
function cargaMedio() {
    numMediosCargados++;
}
```

La función cargaMedio lo único que hará será incrementar una variable global numMediosCargados que habremos declarado previamente inicializándola a 0.

Esto mismo lo haremos con la imagen iFood y con la imagen iWall (ambas llamarán a la misma función cargaMedio en el manejador del evento load).

Por otro lado, tendremos que hacer algo muy parecido con los medios de tipo

Audio:

```
medios['aComer'] = new Audio();
if(canPlayOgg())
        medios['aComer'].src="sounds/chomp.ogg";
else
        medios['aComer'].src="sounds/chomp.m4a";
medios['aComer'].addEventListener("canplaythrough", cargaMedio, false);
```

Como vemos, los sonidos los guardaremos en el mismo array medios, y el manejador del evento canplaythrough llamará a la misma función cargaMedio.El siguiente paso será controlar la carga del juego, de forma que no se inicie hasta que todos los medios estén cargados. Lo único que tendremos que hacer será comprobar en la función iniciar si el número de medios cargados (variable numMediosCargados) es igual a la longitud del array, en cuyo caso, iniciaremos el juego, mientras que, en caso contrario, llamaremos a una función cargando que hará una pequeña pausa y volverá a realizar la comprobación.

El primer problema que nos encontramos es que no vamos a poder obtener la longitud del array medios, ya que se trata de un array asociativo y su propiedad length valdrá 0. Por ello, recurriremos a un pequeño truco: para obtener la longitud del array usaremos el método **getOwnPropertyNames**() (JavaScript 1.8.5).

Dicho método nos devuelve un array escalar con las propiedades (índices) de nuestro objeto, por lo que podemos definir un método longitud que nos devuelva el número de elementos que tiene así:

```
Array.longitud = function(obj) {
return Object.getOwnPropertyNames(obj).length - 1;
}
```

y para obtener la longitud del array haremos esto:

```
longitud = Array.longitud(medios);
```

Con todo esto, la función iniciar quedará como sigue:

```
function iniciar() {
        canvas=document.getElementById('lienzo');
        lienzo=canvas.getContext('2d');
        if (numMediosCargados == Array.longitud(medios)) {
                run();
                repaint();
        } else {
                cargando();
        }
}
```

Sólo nos queda por ver cómo sería la función cargando:

```
function cargando() {
    if (numMediosCargados < Array.longitud(medios)) {
        lienzo.fillStyle="#fff";
        lienzo.fillRect(0,0,canvas.width,canvas.height);
        lienzo.fillStyle="#0f0";
        lienzo.fillText('Cargando ' + numMediosCargados + ' de ' +
        Array.longitud(medios),10,10); setTimeout(cargando, 100);
        }
        else {
        iniciar();
        }
}
```

En esta función, simplemente, se comprueba si el número de medios cargados coincide con la longitud del array, si no es así, se mostrará el texto "cargando" se hará una pausa de 100ms y se volverá a comprobar, en caso

contrario, se llamará a la función iniciar y se iniciará el juego.

Para terminar, tendremos que sustituir en nuestro juego todos los usos de los objetos de las imágenes y los sonidos anteriores por las referencias correspondientes del array medios. Así, donde hacíamos:

```
aComer.play();
```

Ahora haremos:

```
medios['aComer'].play();
```

Y donde hacíamos:

```
lienzo.drawImage(iBody, body[i].x, body[i].y);
```

Ahora haremos:

```
lienzo.drawImage(medios['iBody'], body[i].x, body[i].y);
```

Con esto concluimos el tema de desarrollo de videojuegos con HTML5. A continuación, veremos cómo realizar animaciones sin necesidad de programar nada en JavaScript utilizando para ello CSS3.

2. Animaciones CSS3

Tradicionalmente, cuando queríamos incluir cualquier tipo de animación en nuestras páginas web, teníamos que recurrir a tecnologías como Flash o JavaScript. Si bien la potencia de ambos es descomunal, las nuevas funciones de animación de CSS3 nos facilitan mucho la tarea de crear animaciones sin tener que depender de tecnologías de terceros y sin necesidad de programar.

Ya vimos en el tema 3 como podíamos crear pequeñas animaciones, utilizando para ello las transiciones de CSS3. En este tema, iremos un paso más allá, y encadenaremos múltiples transiciones juntas en la misma propiedad para que sean realizadas una tras otra.

Las animaciones constan de dos componentes: un estilo que describe la animación y un conjunto de fotogramas que indican su estado inicial y final, así como posibles puntos intermedios en la misma.Las animaciones CSS tienen tres ventajas principales sobre las técnicas tradicionales de animación basada en scripts:

1. Es muy fácil crear animaciones sencillas, puedes hacerlo incluso sin tener conocimientos de JavaScript.
2. La animación se muestra correctamente, incluso en equipos poco potentes. Animaciones simples realizadas en JavaScript pueden verse mal (a menos que estén muy bien programadas). El motor de renderizado puede usar técnicas de optimización como el "frame-skipping" u otras para conseguir que la animación se vea tan suave como sea posible.
3. Al ser el navegador quien controla la secuencia de la animación, permitimos que optimice el rendimiento y eficiencia de la misma, por ejemplo, reduciendo la frecuencia de actualización de la animación ejecutándola en pestañas que no estén visibles.

Fotogramas claves de las animaciones CSS3

Para aquellos que hayáis trabajado alguna vez con Flash, el concepto de fotograma clave no necesita explicación. Los fotogramas clave de las animaciones CSS3 son muy similares a los que encontramos en Flash.Un fotograma clave no es más que un punto destacado en el tiempo de nuestra animación. Cada fotograma describe cómo se muestra cada elemento animado en un momento dado durante la secuencia de la animación. Cualquier animación consta al menos de dos fotogramas claves: el punto inicial y el punto final. Imaginad que nuestra animación es como una carretera:

El primer semáforo actuaría como fotograma clave inicial y el segundo como el final. Entre uno y otro se produciría nuestra animación, que no es más que el desplazamiento del coche hacia la derecha.

@keyframes

En CSS3 creamos animaciones completas mediante@keyframes, que son un conjunto de fotogramas clave. Su sintaxis es la siguiente:

```
@keyframes nombreAnimacion{
        puntoDelKeyframe{
                atributosIniciales;
        }
        puntoDelKeyframe{
                nuevosAtributos;
        }
```

```
        .......
        puntoDelKeyframe{
                últimosAtributos;
        }
}
```

Visto así, no queda demasiado claro, pero veamos que tendríamos que hacer para desplazar nuestro coche a la derecha:

```
@keyframes animacionCoche{
        /*Indicamos que salimos de la posición 0*/
        from{
                left:0px; }
        /*Indicamos que al final la posición debe ser 350*/
        to{
                left:350px;
        }
}
```

Los fotogramas usan porcentajes para indicar en qué momento de la secuencia de la animación tienen lugar: 0% es el principio, 100% es el estado final de la animación. Obligatoriamente, debemos especificar estos dos fotogramas para que el navegador sepa dónde debe comenzar y finalizar; debido a su importancia, estos dos fotogramas tienen alias especiales: from y to (ver el ejemplo anterior). Podemos crear animaciones más complejas estableciendo fotogramas claves intermedios mediante porcentajes:

```
@keyframes animacionCoche {
	from {
		left:0px;
	}
	/*Hasta el 65% de la reproducción solo queremos que se desplace 10 píxeles*/
	65%
	{
		left:10px;
	} to {
		left:350px;
	}
}
```

Compatibilidad entre navegadores

Actualmente, para que las animaciones funcionen correctamente en todos los navegadores, debemos utilizar los prefijos propietarios de cada navegador.

```
@keyframes nombreAnimacion { ... }
@-webkit-keyframes nombreAnimacion { ... }
@-moz-keyframes nombreAnimacion { ... }
@-o-keyframes nombreAnimacion { ... }
```

Además, no podremos separar las distintas declaraciones por comas, todas se deben hacer aparte. El siguiente código no funcionaría:

```
@keyframes nombreAnimacion, @-webkit-keyframes nombreAnimacion,
@-moz-keyframes nombreAnimacion, @-o-keyframes nombreAnimacion,
@-ms-keyframes nombreAnimacion { ... }
```

Asignar animaciones CSS3 a un elemento

Para asignar una secuencia de animación CSS3 a un elemento utilizaremos la propiedad **animation** y sus sub-propiedades. Con ellas podemos no sólo configurar el ritmo y la duración de la animación, sino otros detalles sobre la secuencia de la animación.

Con estas propiedades no configuramos la apariencia actual de la animación, para eso disponemos de @keyframes (visto anteriormente). Lo que haremos mediante estas propiedades es asociar una secuencia de animación, creada anteriormente con @keyframes, a un elemento y configurar cómo queremos que se reproduzca la animación.

Las subpropiedades de animation son:

- animation-delay: tiempo de retardo en segundos entre el momento en que el elemento se carga y el comienzo de la secuencia de la animación. animation-direction: indica la dirección de la animación. Los posibles valores son:
  - normal: La animación se reproduce desde el inicio hasta el final.o
  - reverse: La animación se reproduce desde el final hasta el inicio.o
  - alternate: La animación se reproduce normalmente en las iteraciones impares y hacia atrás en las pares.
  - alternate-reverse: La animación se reproduce hacia atrás en las iteraciones impares y hacia delante en las pares.
- animation-duration: indica la cantidad de tiempo que la animación consume en completar su ciclo (duración).
- animation-iteration-count: el número de veces que se repite. Podemos indicar infinite para repetir la animación indefinidamente.
- animation-name: Especifica el nombre de la regla @keyframes que

describe los fotogramas de la animación.

- animation-play-state: permite pausar y reanudar la secuencia de la animación. Se podría usar desde JavaScript para pausar la animación.
- animation-timing-function: indica el ritmo de la animación, es decir, como se muestran los fotogramas de la animación, estableciendo curvas de aceleración. Por defecto, responde al valor ease, pero puede responder a diferentes valores: ease, linear, ease-in, ease-out, ease-in-out.
- animation-fill-mode: especifica qué valores tendrán las propiedades después de finalizar la animación. Los posibles valores son:

none: Es el valor por defecto. El elemento no tendrá ningún estilo aplicado ni antes ni después de la animación.

  - forwards: El elemento se quedará con los estilos aplicados al final de la animación.
  - backwards: El elemento se quedará con los estilos aplicados al principio de la animación.
  - both: El elemento se quedará con los estilos aplicados al principio de la animación y al final de la misma.

Veamos cómo asignaríamos la animación del ejemplo anterior a un elemento de nuestra página que tenga el id coche:

```
#coche {
    animation-duration: 3s;
    animation-name: animacionCoche;
    animation-iteration-count: 1;
    position:relative;
}
```

El estilo del elemento #coche indica, a través de la propiedad animation-duration, que la animación debe durar 3 segundos desde el inicio al fin y que el nombre de los **@keyframes** que definen los fotogramas de la secuencia de la animación es **animacionCoche**.

Además, la animación sólo se reproducirá una vez.

Evidentemente, si quisiéramos que la animación funcione en todos los navegadores, tendremos que utilizar los prefijos correspondientes. Veamos una animación completa, utilizando los prefijos correspondientes:

```
@keyframes animacionCoche {
    from { transform:rotate(-18deg); }
    65% { transform:rotate(18deg); }
    to { transform:rotate(0deg); }
}
@-webkit-keyframes animacionCoche {
    from { -webkit-transform:rotate(-18deg); }
    65% { -webkit-transform:rotate(18deg); }
    to { -webkit-transform:rotate(0deg); }
}
@-moz-keyframes animacionCoche {
    from { -moz-transform:rotate(-18deg); }
    65% { -moz-transform:rotate(18deg); }
    to { -moz-transform:rotate(0deg); }
}
@-o-keyframes animacionCoche {
    from { -o-transform:rotate(-18deg); }
    65% { -o-transform:rotate(18deg); }
    to { -o-transform:rotate(0deg); }
}
```

```
#coche {
    animation-duration: 3s;
    animation-name: animacionCoche;
    animation-iteration-count: 1;

       -webkit-animation-duration: 3s;
       -webkit-animation-name: animacionCoche;
       -webkit-animation-iteration-count: 1;

       -moz-animation-duration: 3s;
       -moz-animation-name: animacionCoche;
       -moz-animation-iteration-count: 1;

       -o-animation-duration: 3s;
       -o-animation-name: animacionCoche;
       -o-animation-iteration-count: 1;

       position:relative;
}
```

En este caso, hemos realizado una animación de rotación para que quede claro que los prefijos hay que utilizarlos tanto al crear la animación como en las propiedades que lo requieran.

Como se puede observar, es bastante tedioso crear una animación que sea compatible con todos los navegadores, pero es el precio que tenemos que pagar en aras de la compatibilidad.

En cualquier caso, pese a este inconveniente, sigue siendo mucho más sencillo crear una animación utilizando CSS3 que utilizando JavaScript.

Controlar la reproducción de las animaciones

Es muy habitual que nos interese reproducir una animación, no cuando se cargue la página, sino como respuesta a un determinado evento. Por ejemplo, nos puede interesar que una animación se inicie cuando colocamos el ratón sobre un elemento. Este caso será el más sencillo de implementar, puesto que disponemos de la pseudo-clase :hover. En el ejemplo anterior, si quisiéramos que el coche se moviera sólo cuando ponemos el ratón sobre él, haríamos lo siguiente:

```
#coche:hover {
        animation-duration: 3s;
        animation-name: animacionCoche;
        animation-iteration-count: 1;

        -webkit-animation-duration: 3s;
        -webkit-animation-name: animacionCoche;
        -webkit-animation-iteration-count: 1;

        -moz-animation-duration: 3s;
        -moz-animation-name: animacionCoche;
        -moz-animation-iteration-count: 1;

        -o-animation-duration: 3s;
        -o-animation-name: animacionCoche;
        -o-animation-iteration-count: 1;

        position:relative;
}
```

Como vemos, este caso es obvio y no supone ninguna complicación, pero ¿qué sucedería si quisiéramos que una animación se iniciara cuando hacemos click con el ratón sobre un elemento? En este caso, no tendríamos más remedio

que recurrir a JavaScript.

Supongamos que queremos que la animación del coche se inicie cuando pulsamos sobre un botón de la página:

```
<button id="iniciaAnimacion">Inicia animación</button>
<div id="coche"><p>coche</p></div>
```

Vamos a utilizar los siguientes estilos:

```
@keyframes animacionCoche {
	from { left: 0px; }
	65% { left: 10px; }
	to { left: 350px; }
}

@-webkit-keyframes animacionCoche {
	from { left: 0px; }
	65% { left: 10px; }
	to { left: 350px; }
}

@-moz-keyframes animacionCoche {
	from { left: 0px; }
	65% { left: 10px; }
	to { left: 350px; }
}

@-o-keyframes animacionCoche {
	from { left: 0px; }
	65% { left: 10px; }
	to { left: 350px; }
}
```

```
.aCoche {
	animation-duration: 3s;
	animation-name: animacionCoche;
	animation-iteration-count: 1;

	-webkit-animation-duration: 3s;
	-webkit-animation-name: animacionCoche;
	-webkit-animation-iteration-count: 1;

	-moz-animation-duration: 3s;
	-moz-animation-name: animacionCoche;
	-moz-animation-iteration-count: 1;

	-o-animation-duration: 3s;
	-o-animation-name: animacionCoche;
	-o-animation-iteration-count: 1;

	position:relative;
}
```

Es importante observar que, en este caso, hemos utilizado un estilo de clase (.aCoche) para asignarle la animación al elemento, por lo tanto, la animación se reproducirá cuando le asociemos la clase al elemento que nos interese. A continuación, vamos a ver el código JavaScript que necesitaremos para asociar la clase aCoche con el elemento coche:

```
function iniciar() {
	var iniciaAnimacion=document.getElementById("iniciaAnimacion");
	iniciaAnimacion.addEventListener("click", accIniciaAnimacion, false);
}
```

```
function accIniciaAnimacion() {
        var coche=document.getElementById("coche");
        coche.className = "aCoche";
}
window.addEventListener("load", iniciar, false);
```

En el momento que pulsamos sobre el botón, se le asignará la clase aCoche al elemento coche, y será en este instante cuando se inicie la animación.

Encadenar animaciones

Hasta ahora hemos visto que las animaciones nos servirán para aplicar varias transiciones consecutivas (una tras otra) sobre un mismo elemento. Muy bien, pero ¿y si quisiéramos aplicar varias transiciones consecutivas, pero no sobre un mismo elemento, sino sobre elementos diferentes? La respuesta es obvia, tendríamos que crear una animación diferente para cada elemento que quisiéramos animar. Perfecto, pero también hemos visto que las animaciones se iniciarán en el momento que se asocie el estilo que contiene la animación al elemento que queremos animar, entonces, ¿cómo haríamos para encadenar animaciones?, es decir, ¿cómo podemos hacer para que se inicie una animación en el momento que finalice la anterior? para hacer esto se han incorporado nuevos eventos para el control de las animaciones. En concreto, disponemos de tres eventos que nos darán información sobre el estado en el que se encuentra la animación. Estos eventos son los siguientes:

- **animationstart**: este evento será lanzado cuando se inicie la animación por primera vez. Si la animación está configurada para hacer varias iteraciones, sólo se lanzará el evento cuando se inicie la primera de ellas.

- **animationiteration**: este evento será lanzado al inicio de cada iteración de la animación, excepto en la primera.
- **animationend**: este evento será lanzado al final la animación.

Con esto, ya podemos encadenar nuestras animaciones, por ejemplo, veamos cómo haríamos para que nuestro coche se desplace a la derecha hasta chocar con otro coche y, en ese momento, el otro coche inicie otro desplazamiento a la derecha.

```
<button id="iniciaAnimacion">Inicia animación</button>
<div id="coche1"><p>coche1</p></div>
<div id="coche2"><p>coche2</p></div>
```

Ahora tenemos dos coches en lugar de uno. Los estilos iniciales serán los siguientes:

```
#coche1 {
        position:absolute;
        top:200px;
        left: 0px;
        width: 50px;
        display:inline;
}

#coche2 {
        position:absolute;
        top:200px;
        left: 400px;
        width: 50px;
        display:inline;
}
```

A continuación, veremos los estilos que crearemos para las animaciones:

```
@keyframes animacionCoche1 {
    from { left: 0px; }
    65% { left: 10px; }
    to { left: 350px; }
}

@-webkit-keyframes animacionCoche1 {
    from { left: 0px; }
    65% { left: 10px; }
    to { left: 350px; }
}

@-moz-keyframes animacionCoche1 {
from { left: 0px; }
    65% { left: 10px; }
    to { left: 350px; }
}

@-o-keyframes animacionCoche1 {
    from { left: 0px; }
    65% { left: 10px; }
    to { left: 350px; }
}
```

```
.aCoche1 {
        animation-duration: 3s;
        animation-name: animacionCoche1;
        animation-iteration-count: 1;
        animation-fill-mode: forwards;

        -webkit-animation-duration: 3s;
        -webkit-animation-name: animacionCoche1;
        -webkit-animation-iteration-count: 1;
         -webkit-animation-fill-mode: forwards;

        -moz-animation-duration: 3s;
        -moz-animation-name: animacionCoche1;
        -moz-animation-iteration-count: 1;
        -moz-animation-fill-mode: forwards;

        -o-animation-duration: 3s;
        -o-animation-name: animacionCoche1;
        -o-animation-iteration-count: 1;
        -o-animation-fill-mode: forwards;
}

@keyframes animacionCoche2 {
   from { left: 400px; }
   65% { left: 550px; }
   to { left: 600px; }
}
```

```
@-webkit-keyframes animacionCoche2 {
    from { left: 400px; }
    65% { left: 550px; }
    to { left: 600px; }
}

@-moz-keyframes animacionCoche2 {
    from { left: 400px; }
    65% { left: 550px; }
    to { left: 600px; }
}

@-o-keyframes animacionCoche2 {
    from { left: 400px; }
65% { left: 550px; }
    to { left: 600px; }
}

.aCoche2 {
      animation-duration: 3s;
      animation-name: animacionCoche2;
      animation-iteration-count: 1;
      animation-fill-mode: forwards;

      -webkit-animation-duration: 3s;
      -webkit-animation-name: animacionCoche2;
      -webkit-animation-iteration-count: 1;
      -webkit-animation-fill-mode: forwards;

      -moz-animation-duration: 3s;
      -moz-animation-name: animacionCoche2;
      -moz-animation-iteration-count: 1;
      -moz-animation-fill-mode: forwards;
```

```
        -o-animation-duration: 3s;
        -o-animation-name: animacionCoche2;
        -o-animation-iteration-count: 1;
        -o-animation-fill-mode: forwards;
}
```

Finalmente, veremos el código JavaScript necesario para encadenar las animaciones:

```
function iniciar() {
        var iniciaAnimacion=document.getElementById("iniciaAnimacion");
        iniciaAnimacion.addEventListener("click", accIniciaAnimacion, false);
}

function accIniciaAnimacion() {
        var coche1=document.getElementById("coche1");

        coche1.className = "aCoche1";
        coche1.addEventListener("animationend", animacionCoche1Fin, false);
}

function animacionCoche1Fin() {
        var coche2=document.getElementById("coche2");
        coche2.className = "aCoche2";
}

window.addEventListener("load", iniciar, false);
```

Como vemos en el ejemplo, esperamos a que termine la primera animación para iniciar la siguiente.

Compatibilidad entre navegadores

Como siempre, el problema que nos encontramos es la compatibilidad entre navegadores. Aunque estos eventos ya forman parte del estándar W3C, todavía tenemos que utilizar prefijos (al estilo de CSS3) para asegurarnos del correcto funcionamiento en todos los navegadores modernos. En la siguiente tabla podemos ver los diferentes prefijos utilizados para cada uno de los navegadores.

W3C standard	Firefox	webkit	Opera	IE10
animationstart	mozAnimationStart	webkitAnimationStart	oanimationstart	MSAnimationStart
animationiteration	mozAnimationIteration	webkitAnimationIteration	oanimationiteration	MSAnimationIteration
animationend	mozAnimationEnd	webkitAnimationEnd	oanimationend	MSAnimationEnd

Para que nuestro código funcione correctamente en todos los navegadores, tendremos que introducir un manejador de evento para cada uno de los diferentes eventos existentes:

```
coche1.addEventListener("animationend", animacionCoche1Fin, false);
coche1.addEventListener("webkitAnimationEnd", animacionCoche1Fin, false);
coche1.addEventListener("mozAnimationEnd", animacionCoche1Fin, false);
coche1.addEventListener("oanimationend", animacionCoche1Fin, false);
coche1.addEventListener("MSAnimationEnd", animacionCoche1Fin, false);
```

De esta forma, el evento que reconozca el navegador será manejado y el resto serán ignorados. Como vemos, es bastante tedioso tener que añadir todos estos eventos, por lo que, utilizaremos una función que lo haga por nosotros:

```
function PrefixedEvent(element, type, callback) {
        for (var p = 0; p < pfx.length; p++) {
                if (pfx[p] == "" || pfx[p] == "o")
                        type = type.toLowerCase();
                element.addEventListener(pfx[p]+type, callback, false);
        }
}
```

Esta función generara los diferentes manejadores de evento para todas las posibilidades que existen. Cada vez que tengamos que crear un manejador de evento llamaremos a esta función. Por ejemplo, la llamada anterior quedaría así:

```
PrefixedEvent(coche1, "AnimationEnd", animacionCoche1Fin);
```

Siempre pasaremos el nombre del evento con las iniciales de las palabras en mayúsculas, la función ya se encargará de convertirlas a minúsculas cuando sea necesario. Utilizando esta función, el ejemplo anterior quedaría de la siguiente forma:

```
function iniciar() {
        var iniciaAnimacion=document.getElementById("iniciaAnimacion");
        iniciaAnimacion.addEventListener("click", accionIniciaAnimacion,
        false);
}

function accionIniciaAnimacion() {
        var coche1=document.getElementById("coche1");

        coche1.className = "aCoche1";
        PrefixedEvent(coche1, "AnimationEnd", animacionCoche1Fin);
}
```

```
function animacionCoche1Fin() {
	var coche2=document.getElementById("coche2");

	coche2.className = "aCoche2";
}

window.addEventListener("load", iniciar, false);

var pfx = ["webkit", "moz", "MS", "o", ""];

function PrefixedEvent(element, type, callback) {
	for (var p = 0; p < pfx.length; p++) {
		if (pfx[p] == "" || pfx[p] == "o")
			type = type.toLowerCase();
		element.addEventListener(pfx[p]+type, callback, false);
	}
}
```

Podéis encontrar este ejemplo completamente funcional en la carpeta ejemplo- animacion-coche del fichero de recursos.

Con esto, damos por finalizados los contenidos del tema. A continuación, plantearemos una serie de ejercicios para practicar todo lo estudiado.

3. Ejercicios

Ejercicio 1

En este primer ejercicio, vamos a implementar el videojuego de la serpiente que se ha explicado a lo largo del tema. Para ello, crearemos una nueva página en nuestro sitio web del curso llamada serpiente.html. La página tendrá los mismos contenidos que el resto de páginas del sitio web, a excepción de la sección <main>, que contendrá un con id **contenedorVideojuego** y, dentro de este, tendremos el <canvas> sobre el que implementaremos nuestro videojuego.

El <canvas> tendrá un ancho de 500 píxeles y un alto de 300 píxeles y le pondremos el id lienzo.

Crearemos una hoja de estilos para esta sección que llamaremos videojuego.css.

En esta hoja de estilos tendremos las siguientes reglas de estilo:

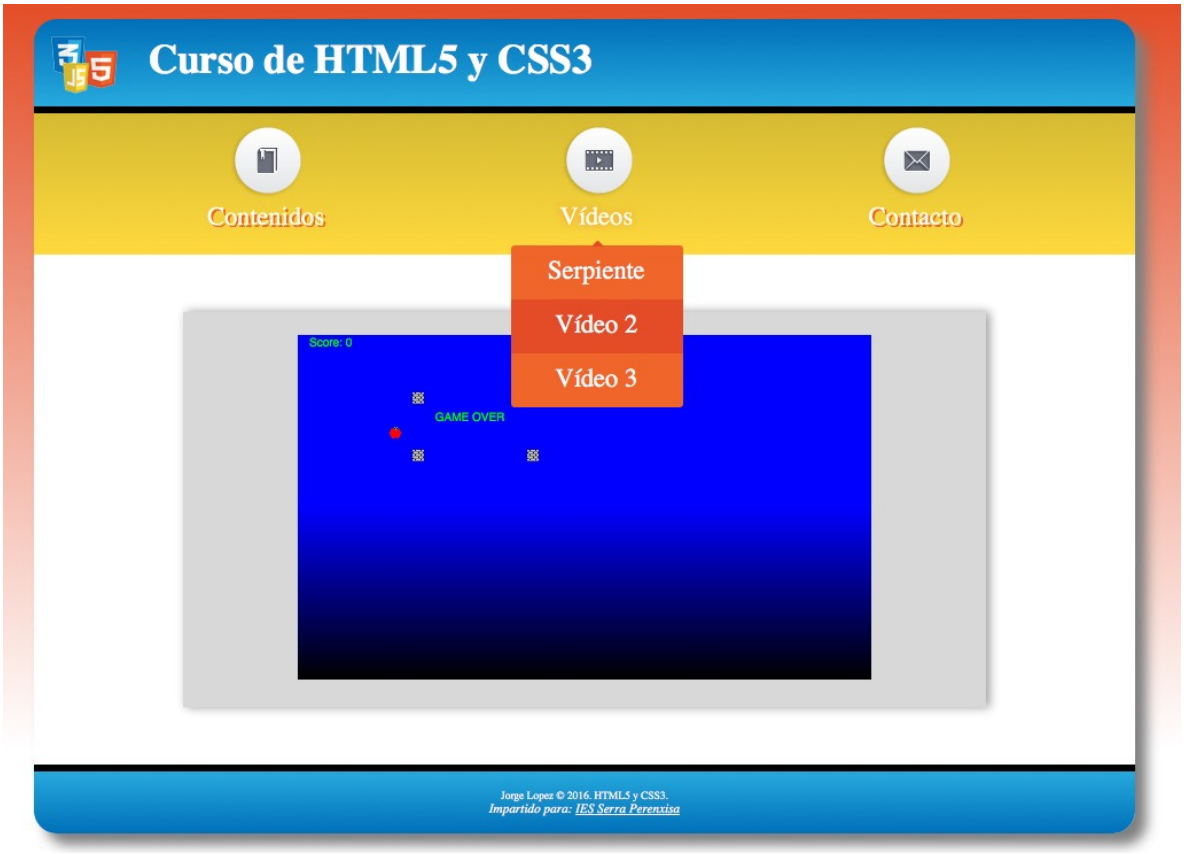
```
main {
        text-align:center;
}

#contenedorVideojuego {
        display: inline-block;
        margin:50px;
        width:600px;
        padding:20px 50px;
        background: rgba(200, 200, 200, 0.7);
        text-align:center;
```

```
        -webkit-box-shadow:3px 0px 8px rgba(0, 0, 0, 0.4);
        -moz-box-shadow:3px 0px 8px rgba(0, 0, 0, 0.4);
        -o-box-shadow:3px 0px 8px rgba(0, 0, 0, 0.4);
        box-shadow:3px 0px 8px rgba(0, 0, 0, 0.4);
}
```

Para la implementación del videojuego, crearemos un fichero JavaScript que llamaremos serpiente.js. Evidentemente, tendremos que enlazar el fichero JavaScript y la hoja de estilos con el documento html.

Todas las imágenes y sonidos necesarios para el videojuego los podéis encontrar en el fichero de recursos. Las imágenes las guardaremos en la carpeta imgs, mientras que los sonidos los guardaremos en una carpeta que llamaremos sounds. Para acceder al documento serpiente.html, cambiaremos el primer enlace del menú desplegable que creamos en el tema 3. El texto del enlace será serpiente y el href apuntará a serpiente.html.

Cambiaremos el ancho del menú a 150px (recordad ajustar los márgenes para que el menú quede centrado). Esto deberemos cambiarlo en los demás documentos de nuestro sitio web: index.html, videos.html, contacto.html y, por supuesto, en serpiente.html.El aspecto de la página serpiente.html será el siguiente:



En el fichero de recursos tenéis la imagen serpiente.jpg para ver el resultado con mejor resolución.

Ejercicio 2

En este segundo ejercicio, haremos una pequeña modificación sobre el videojuego de la serpiente. En concreto, lo que tenemos que hacer es que las paredes se muevan, de forma que, si chocan con la cabeza o con cualquier otra parte del cuerpo de la serpiente, el juego finalice.

El movimiento de las paredes será aleatorio, para lo cual podemos utilizar la función random de la siguiente manera:

```
var dirWall = random(4);
```

Con la instrucción anterior, obtendremos un número aleatorio entre 0 y 3, que se corresponde con los valores que toman las constantes que definimos

para las direcciones de movimiento.

En función de la dirección aleatoria obtenida desplazaremos la pared 10 píxeles a la izquierda, derecha, arriba o abajo.Evidentemente, esto lo haremos dentro de un bucle para que el movimiento de cada pared sea independiente del resto.

Otra cosa que debemos tener en cuenta es que ahora las paredes no sólo podrán chocar con la cabeza, también podrán chocar con cualquier parte del cuerpo de la serpiente.

Ejercicio 3

En este ejercicio trabajaremos una animación css3 sencilla. Vamos a animar el logo de nuestra cabecera, de forma que estará siempre rotando. Para ello, crearemos una animación con tres keyframes: uno inicial que tendrá rotación de 0o, otro en el 50% que tendrá una rotación sobre el eje y de 180o y uno final que volverá a tener una rotación de 0o. Evidentemente, utilizaremos las transformaciones de rotación que vimos en el tema 3.

La animación se la aplicaremos a la imagen del logo que tenemos en la cabecera, le daremos una duración de 5 segundos y le indicaremos que se repita de forma indefinida.

Ejercicio 4

Para terminar, haremos una serie de animaciones encadenadas que controlaremos mediante JavaScript. Para ello, crearemos un fichero JavaScript que llamaremos animación.js y lo asociaremos al documento index.html.Los elementos que animaremos serán los siguientes: las tres chinchetas que colocamos sobre los móviles y los dos móviles que aparecen inclinados.

A continuación, veremos cómo serán estas animaciones y en qué orden se reproducirán:

1. El primer elemento que animaremos será la chincheta que aparece en el primer <article>. La animación se llamará mueveChincheta1 y desplazará la chincheta desde la posición 0px (propiedad left) hasta la posición 199px. Comenzará cuando se cargue la página y la controlaremos mediante JavaScript, es decir, crearemos un estilo de clase (chincheta1) que asociaremos con el que se encuentra dentro del primer <article> en el evento load de la página (podéis utilizar el método querySelector que vimos en el tema 3).

   Los estilos que aplicaremos a la clase chincheta1 serán los mismos que teníamos para la clase chincheta del tema anterior, más la propiedad left:0px. Además, le añadiremos los estilos necesarios para asignarle la animación mueveChincheta1 que durará un segundo y mantendrá la posición que haya al final de la animación.
2. A continuación, animaremos el primer <article> de nuestra página. La animación se llamará animArticle1 y rotará el elemento desde los 0o hasta los -2o. Comenzará cuando finalice la anterior, durará medio segundo y mantendrá la rotación que haya al final. La animación la asociaremos mediante un estilo de clase llamado article1.
3. En tercer lugar, animaremos la chincheta del segundo <article>. La animación se llamará mueveChincheta2 y desplazará la chincheta desde la posición 0px (propiedad left) hasta la posición 120px. Comenzará cuando finalice la anterior, durará un segundo y mantendrá la posición que haya al final.

   La animación la asociaremos mediante un estilo de clase llamado

chincheta2 que tendrá los mismos estilos que chincheta1, a excepción de la animación asociada.

4. Después, animaremos la chincheta del tercer <article>. La animación se llamará mueveChincheta3 y desplazará la chincheta desde la posición 0px (propiedad left) hasta la posición 30px. Comenzará cuando finalice la anterior, durará un segundo y mantendrá la posición que haya al final.

   La animación la asociaremos mediante un estilo de clase llamado chincheta3 que tendrá los mismos estilos que chincheta1, a excepción de la animación asociada.

5. Por último, animaremos el tercer <article> de nuestra página. La animación se llamará animArticle3 y rotará el elemento desde los 0o hasta los 2o. Comenzará cuando finalice la anterior, durará medio segundo y mantendrá la rotación que haya al final. La animación la asociaremos mediante un estilo de clase llamado article3.

Con esto habremos terminado los ejercicios del tema. Para facilitar la comprensión de los mismos podéis encontrar un vídeo en el que podéis ver cómo debe quedar nuestro sitio web cuando finalice este tema.

Recuerda que para realizar la entrega del ejercicio, debes comprimir la carpeta curso dentro de un fichero llamado:

nombre-alumno-tema5.zip

No olvidéis pasar el validador tanto a los html como a los css antes de efectuar la entrega.