

# Juegos Canvas Ninja

Crea Juegos en HTML5 Canvas como un Ninja

## ¡Bienvenido!

Este pequeño curso está destinado para gente que ya tenga conocimientos de programación y/o Javascript. En caso que no se tengan conocimientos o se desee más profundizar en el lenguaje, recomendamos el curso en línea de [javascriptya.com.ar](http://javascriptya.com.ar), donde se aprenderá desde el comienzo y de forma muy sencilla dicho lenguaje.

Crear juegos en canvas es sencillo y no requiere de programas especializados o costosos. Puedes editar los archivos con cualquier editor de textos como Notepad (block de notas de Windows). Aun así, recomendamos usar mejor Notepad++, ya que su habilidad de colorear sintaxis, ayuda a leer mejor los códigos y cometer menos errores. Puedes descargarlo en [notepad-plus-plus.org](http://notepad-plus-plus.org). Si deseas un editor más robusto, con sugerencias de autocompletado y detección de posibles errores, puedes probar [brackets.io](http://brackets.io)

Ahora con estas herramientas ¡Ya estás listo para crear tus propios juegos!

## Índice



### 1. Mi primer juego

- 1.1. Dibujando en el Canvas.
- 1.2. Animando el Canvas.
- 1.3. Usando el teclado.
- 1.4. Interactuando con otros elementos.
- 1.5. Interactuando con varios elementos iguales.
- 1.6. Imágenes y sonidos.
- 1.7. Optimización para Javascript.
- Apéndice 1: RequestAnimationFrame.
- Apéndice 2: RequestAnimationFrame: Regulando el tiempo entre dispositivos.
- Apéndice 3: Programación Orientada a Objetos.



### Extras

- Estirar el canvas y llenar la pantalla.
- Doble búfer y escalado pixelado.
- Manejo de escenas.
- Almacenamiento local y altos puntajes.



### 2. Un juego de naves

Requiere: Conocimientos básicos.

- 2.1. Mover mientras se presiona tecla.
- 2.2. Municiones dinámicas.
- 2.3. Naves Enemigas.
- 2.4. Vida y Daño.
- 2.5. Mejoras.
- 2.6. Hojas de Sprites y animaciones.
- Apéndice 1. Estrellas.
- Apéndice 2. Fondo en movimiento.
- Enemigos comunes: El disparador.
- Enemigos comunes: El disparador de 8 lados.



### 3. Usando el ratón

Requiere: Conocimientos básicos.

- 3.1. Usando el ratón.
- 3.2. Distancia entre círculos.
- 3.3. Presionando el botón del ratón.
- 3.4. Midiendo el tiempo.
- 3.5. Dispara al objetivo.
- 3.6. Desplazamiento angular.
- 3.7. ¡Huye!
- Extra: Sistema de partículas.



### 4. Otro juego de naves

Requiere: Tema 2. Un juego de naves, Tema 3. Usando el ratón.

- 4.1. Aceleración.
- 4.2. Rotación de imágenes.
- 4.3. Animación condicional.
- 4.4. Asteroides fragmentables.
- 4.5. Explosión y regeneración.
- 4.6. Oleadas de asteroides.



## 5. Arrastra y suelta

Requiere: Tema 3. Usando el ratón, Tema 4. Otro juego de naves.

- 5.1. Arrastra y suelta
- 5.2. Soporte básico a dispositivos multi-toque
- Ejercicio: Arrastra al agujero
- 5.3. Rectángulos y el ratón
- 5.4. Getters & Setters
- 5.5. Ajusta a la rejilla
- 5.6. Transición de animaciones
- Apéndice 1. Clic secundario y rotación opuesta.
- Apéndice 2. Reordenar un arreglo al azar.
- Reto: Rompecabezas animado.



## 6. Caminando entre laberintos

Requiere: Tema 5. Arrastra y suelta.

- 6.1. Objetos sólidos.
- 6.2. Mapas de mosaico.
- 6.3. Mundos grandes y cámara.
- 6.4. Transición de mapas.
- 6.5. Paredes y enemigos.
- 6.6. Gráficos direccionales.
- 6.7. Proyección Isométrica.



## 7. Saltando a las plataformas

Requiere: Tema 6. Caminando entre laberintos.

- 7.1. Gravedad.
- 7.2. Saltando a las plataformas.
- 7.3. Volteando imágenes.
- Apéndice 1. Mapas con colindancia automatizados.
- Reto: Terrenos con laderas.
- (En curso)



## 8. Juegos para móviles

Requiere: Tema 2. Un juego de naves.

- 8.1. Multi-toque.
- 8.2. Pantalla completa en dispositivos móviles.
- 8.3. Botones en pantalla.
- 8.4. Acelerómetro.



## 9. MMOG: Juegos Multijugador Masivos en Línea

Requiere: Tema 3. Usando el ratón.

- 9.1. Introducción a Node.js.
- 9.2. Introducción a Socket.io.
- 9.3. Disparen al objetivo.
- Apéndice 1. Creando un servidor.



## Tips para artistas

- Creando imágenes de mosaico.
- Creando sprites para un juego.

## 01.01. Snake - Dibujando en el canvas

Para crear un juego en canvas, necesitaremos dos archivos: Una página HTML para mostrarlo, y el código en JavaScript del juego. Comencemos creando un archivo llamado "index.html", lo abrimos para editar (Clic derecho » Abrir con » Notepad++, o el que uses) y copiamos dentro el siguiente código:

```
1 | <!DOCTYPE html>
2 | <html lang="es">
3 |   <head>
4 |     <meta charset="UTF-8" />
5 |     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6 |     <!--[if lte IE 8]><meta http-equiv="X-UA-Compatible" content="chrome=1" /><![endif]-->
7 |     <title>My First Canvas Game</title>
8 |   </head>
9 |
10 |   <body>
11 |     <h1>My First Canvas Game</h1>
12 |
13 |     <p><canvas id="canvas" width="300" height="150" style="background:#999">
14 |       Canvas not supported by your browser.
15 |     </canvas></p>
16 |
17 |     <script type="application/javascript" src="game.js"></script>
18 |   </body>
19 | </html>
```

markup

No me detendré a explicar el código HTML, ya que no es necesario para comprender el desarrollo de juegos. Digamos que el código arriba es lo básico para mostrar una página web donde mostraremos nuestro juego. Solo hay dos líneas importantes que debes comprender:

```
17 |   <script type="application/javascript" src="game.js"></script>
```

markup

Mediante esa etiqueta "script", llamamos al archivo game.js, que contendrá todo el código de nuestro juego.

```
13 |   <canvas id="canvas" width="300" height="150" style="background:#999">
14 |     Canvas not supported by your browser.
15 |   </canvas>
```

markup

En esta etiqueta canvas, será donde dibujaremos nuestro juego, el ancho de 300px y alto de 150px son los valores predeterminados, por lo que se creará un lienzo del mismo tamaño en caso de no especificarse estos atributos. Puedes personalizarlos de acuerdo al juego que desarrolles, mas para este ejemplo, los dejaremos así.

El ID es el nombre único de nuestro elemento en la página, y es necesario para hacer referencia a él desde el juego; podemos cambiarlo al que deseemos, pero necesitaremos también cambiarlo en nuestro código (Que veremos en un momento).

Por último, agregamos un fondo gris para identificar dónde se encuentra nuestro lienzo (canvas). Si hay un problema con tu código, únicamente se mostrará el fondo gris, funcionando esto como una alerta. Posteriormente podrás borrarlo o cambiarlo, pero por ahora, será mejor dejarlo ahí.

Ahora, crearemos el código para nuestro juego. Comenzaremos por lo más sencillo: Dibujar un rectángulo dentro de él. Para ello, crearemos un segundo archivo de nombre "game.js", y copiaremos el siguiente código dentro:

```
1 | var canvas = null,
2 |   ctx = null;
3 |
4 | function paint(ctx) {
5 |   ctx.fillStyle = '#0f0';
6 |   ctx.fillRect(50, 50, 100, 60);
7 | }
8 |
9 | function init() {
10 |   canvas = document.getElementById('canvas');
11 |   ctx = canvas.getContext('2d');
12 |   paint(ctx);
13 | }
14 |
15 | window.addEventListener('load', init, false);
```

javascript

Analizaremos el código por partes, para hacerlo más entendible, empezaremos por el final.

En la última línea, se agrega un escucha a la ventana, para que en cuanto termine de cargar la página, comience a ejecutar "init" (Que es donde comenzamos nuestro código). Es importante que indiquemos que el código comience hasta que se cargue la página, de lo contrario, el código podría no encontrar nuestro lienzo, y generaría una serie de errores que no nos permitirían reproducir nuestro juego.

Posteriormente en el bloque anterior, empezamos con la función "init". En la primer línea de su contenido se obtiene el lienzo, buscándolo por su ID "canvas" (Si pusiste otro nombre a tu lienzo, es aquí donde debes poner el mismo nombre). Después de esto, se obtiene el contexto 2D de dicho lienzo. Este contexto es necesario ya que es nuestra herramienta para pintar dentro del lienzo; podríamos imaginar que es como nuestro pincel. Por último, se llama a la función "paint", al que se le pasa dicho contexto para dibujar en él.

El siguiente bloque superior es la función "paint". Aquí se indica que será usado el color hexadecimal de relleno '#0f0' (verde), y debajo, se rellenará un rectángulo desde la coordenada x,y 50,50, con 100 de ancho y 60 de alto.

Finalmente, al comienzo se crea las dos variables nulas donde se guardará el lienzo y su contexto.

Guardemos el archivo. Si lo hemos hecho todo bien, al hacer doble clic en index.html, se abrirá una página web donde se mostrará el lienzo con el rectángulo *verde* que hemos creado.

Diviértete cambiando los colores y dibujando más rectángulos, hasta que te familiarices con el lienzo. Puedes usar `strokeStyle` y `strokeRect` para dibujar el contorno en lugar de rellenarlos.

### Código final:

javascript

```
1  var canvas = null,
2      ctx = null;
3
4  function paint(ctx) {
5      ctx.fillStyle = '#0f0';
6      ctx.fillRect(50, 50, 100, 60);
7  }
8
9  function init() {
10     canvas = document.getElementById('canvas');
11     ctx = canvas.getContext('2d');
12     paint(ctx);
13 }
14
15 window.addEventListener('load', init, false);
```

### Depuración

Si solo se muestra en la pantalla un rectángulo gris, probablemente hay un error en tu código. Para encontrar el origen del problema, debes depurarlo.

Dependiendo del navegador que uses, puedes depurar tu código como se muestra a continuación:

#### Chrome:

Presiona F12 o Ctrl + Shift + I y selecciona la pestaña "consola".

#### Firefox:

Presiona Ctrl + Shift + K y selecciona la pestaña "Consola web". Te recomiendo deshabilitar todas las notificaciones excepto las de JS para depurar mas fácil el código de tu videojuego.

#### Internet Explorer:

Presiona F12 y selecciona la pestaña "consola".

#### Opera:

Presiona Ctrl + Shift + I y selecciona la pestaña "consola".

#### Safari:

Presiona Ctrl + Shift + I y selecciona la pestaña "consola". Debes habilitar antes "Mostrar menú de desarrollo en la barra de menú" en la pestaña "Avanzados" en "Preferencias" para usar esta opción.

Con esta información, podrás resolver cualquier problema que puedas encontrarte mientras estás desarrollando tus juegos.

## 01.02. Snake - Animando el canvas.

En la primer parte, vimos como dibujar en nuestro lienzo. Eso está bien, pero un juego se trata de interactuar con los objetos, no solo de dibujarlos. Por tanto, necesitamos por comenzar a darle movimiento a nuestro rectángulo.

Primero, declararemos dos nuevas variables: "x" y "y". Al comienzo, tras declarar las variables canvas y ctx, agregaremos la siguiente línea:

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

javascript

Y modificaremos nuestra función "paint" para que limpie la pantalla antes de volver a dibujar en ella, esto se hace dibujando un rectángulo del tamaño completo del lienzo. Posteriormente dibujamos el rectángulo en las coordenadas mencionadas, el cual haremos de paso un poco más pequeño:

```
function paint(ctx) {
    ctx.fillStyle = '#000';
    ctx.fillRect(0, 0, canvas.width, canvas.height);

    ctx.fillStyle = '#0f0';
    ctx.fillRect(x, y, 10, 10);
}
```

javascript

En este ejemplo, estamos dibujando el rectángulo de fondo de color negro. Puedes darle cualquier valor hexadecimal desde '#000' hasta '#fff'. Experimenta con colores diferentes para que encuentres el que consideres quede mejor con tu juego.

Ahora, nuestra función "init" llama solo una vez a la función "paint", por lo que todo es pintado solo una vez. Para que en verdad se trate de una animación, tenemos que hacer que se llame a la función una y otra vez cada determinado tiempo. Para esto, llamaremos a una función "run()", sustituyendo donde se llamaba a "paint(ctx)" en la función "init":

```
run();
```

javascript

Y crearemos la función "run" de esta forma:

```
function run() {
    window.requestAnimationFrame(run);
    act();
    paint(ctx);
}
```

javascript

En la primer línea, llamaremos a un requestAnimationFrame. Esta función pedirá al navegador para el siguiente momento en que pueda realizar un cuadro de animación, usualmente a 60 cuadros por segundo en computadoras de buen rendimiento, aunque puede variar en computadoras de rendimiento menor. Para saber más al respecto, ve al [Apendice 1: RequestAnimationFrame](#).

Posteriormente, llamamos a las funciones "act()" y "paint(ctx)". Hemos visto antes que "paint" dibuja todo en nuestro lienzo. La función "act" es usada para llamar a todas las acciones en nuestro juego; en este caso, moveremos nuestro rectángulo sumándole 2 pixeles por cuadro a nuestra variable "x":

```
function act(){
    x += 2;
}
```

javascript

Aun cuando las acciones pudieran ser puestas en la función "paint", es recomendado hacerlo por aparte, para prevenir errores, como en el que lo que se muestra en pantalla no es lo mismo que ocurre realmente (Si mezclas ambos al mismo tiempo esto puede ocurrir, por eso es mejor mover primero y al final dibujarlo). Además, hacerlo de forma separada facilita saber dónde ocurren los eventos para modificarlos posteriormente.

Guardemos y abramos de nuevo nuestra página "index.html". Si lo hicimos bien, ¡Veremos a nuestro pequeño rectángulo correr por el lienzo! Allá va... Allá va... Y se fue...

Sí, se fue, y no volverá... Si queremos verlo de nuevo, habrá que actualizar la página, y volverá a hacer lo mismo. Pero si queremos mantener a nuestro rectángulo dentro, podemos condicionarlo a que regrese a la pantalla si sale de esta, agregando las siguientes dos líneas después de "x += 2;":

```
if (x > canvas.width) {
    x = 0;
}
```

javascript

Así le indicamos que si su posición es mayor a la del ancho de lienzo, regrese a la posición 0. Si actualizamos la página ahora, veremos que nuestro rectángulo sale por una esquina, y vuelve a aparecer una y otra vez por la opuesta.

### Problema de compatibilidad:

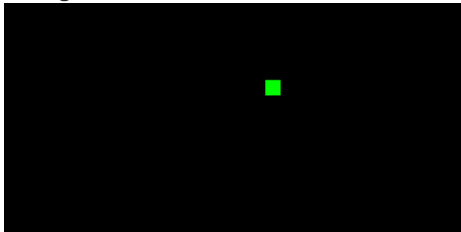
Debido a que algunas versiones antiguas de navegadores no soportan requestAnimationFrame como tal, deberías agregar esta función al final de

tu código (más información en el [Apéndice 1: RequestAnimationFrame](#)):

javascript

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

**Código final:**



javascript

```
1  var canvas = null,
2      ctx = null,
3      x = 50,
4      y = 50;
5
6  window.requestAnimationFrame = (function () {
7      return window.requestAnimationFrame ||
8          window.mozRequestAnimationFrame ||
9          window.webkitRequestAnimationFrame ||
10         function (callback) {
11             window.setTimeout(callback, 17);
12         };
13 }());
14
15 function paint(ctx) {
16     ctx.fillStyle = '#000';
17     ctx.fillRect(0, 0, canvas.width, canvas.height);
18
19     ctx.fillStyle = '#0f0';
20     ctx.fillRect(x, y, 10, 10);
21 }
22
23 function act() {
24     x += 2;
25     if (x > canvas.width) {
26         x = 0;
27     }
28 }
29
30 function run() {
31     window.requestAnimationFrame(run);
32     act();
33     paint(ctx);
34 }
35
36 function init() {
37     canvas = document.getElementById('canvas');
38     ctx = canvas.getContext('2d');
39
40     run();
41 }
42
43 window.addEventListener('load', init, false);
```

## 01.03. Snake - 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;
```

javascript

Y agregar al final de nuestro código un escucha del teclado que almacene la tecla presionada:

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

javascript

Mediante este método, 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. Una buena forma de saber cuál ha sido la última tecla presionada, sería agregando esta línea en nuestra función "paint":

```
ctx.fillText('Last Press: ' + lastPress, 0, 20);
```

javascript

Por ahora no debes preocuparte de eso. Usaremos las teclas izquierda, arriba, derecha y abajo, cuyos valores numéricos son 37, 38, 39 y 40 respectivamente. Para usarlas más fácilmente, las guardaremos en valores constantes:

```
var KEY_LEFT = 37,  
    KEY_UP = 38,  
    KEY_RIGHT = 39,  
    KEY_DOWN = 40;
```

javascript

A diferencia de otros lenguajes de programación, JavaScript no tiene constantes, pero las variables se encargarán del trabajo sin problema.

Vayamos ahora al movimiento del rectángulo. Primero, necesitaremos una nueva variable que almacene la dirección de nuestro rectángulo:

```
var dir = 0;
```

javascript

Esta variable "dir" tendrá un valor del 0 al 3, siendo 0 hacia arriba, y rotando en dirección de las manecillas del reloj para demás valores cada cuarto de hora.

Ahora utilizaremos un método simple para tener un tiempo consistente entre dispositivos (Puedes leer más en el [Apéndice 2: Tiempo consistente entre dispositivos](#)). La forma más fácil para hacer esto por ahora, es dividir las funciones act y paint en dos llamadas distintas, una optimizada para el re-pintar, y una regulada para las acciones:

```
function repaint() {  
    window.requestAnimationFrame(repaint);  
    paint(ctx);  
}  
  
function run() {  
    setTimeout(run, 50);  
    act();  
}
```

javascript

Como puedes ver dentro de la función "run", llamamos a un temporizador "setTimeout", que llama a la función "run" de nuevo cada 50 milisegundos. Esta es una forma simple de tener el juego a 20 ciclos por segundo.

Nota que, dado que este hace el ciclo de la función "act" asíncrono de la función "repaint", no puedes confiar que lo que realices en el primero, será dibujado en el segundo; pero esto no suele ser un problema cuando el re-pintado es más rápido que el ciclo de acciones. ¡No olvides llamar a ambas funciones en la función init!

Ahora comencemos por detectar la dirección que tomará nuestro rectángulo dependiendo la última tecla presionada, dentro de la función "act":

```
// Change Direction  
if (lastPress == KEY_UP) {  
    dir = 0;  
}  
if (lastPress == KEY_RIGHT) {  
    dir = 1;  
}  
if (lastPress == KEY_DOWN) {  
    dir = 2;  
}  
if (lastPress == KEY_LEFT) {
```

javascript

```

    dir = 3;
}

```

Después, moveremos nuestro rectángulo dependiendo la dirección que se haya tomado:

javascript

```

// Move Rect
if (dir == 0) {
    y -= 10;
}
if (dir == 1) {
    x += 10;
}
if (dir == 2) {
    y += 10;
}
if (dir == 3) {
    x -= 10;
}

```

Por último, buscaremos si el rectángulo ha salido de la pantalla, y en dado caso, lo regresaremos a la misma:

javascript

```

// Out Screen
if (x > canvas.width) {
    x = 0;
}
if (y > canvas.height) {
    y = 0;
}
if (x < 0) {
    x = canvas.width;
}
if (y < 0) {
    y = canvas.height;
}

```

Te habrás dado cuenta que antes de cada bloque de código, agregué una referencia precedida de dos diagonales (//). Esto es un comentario, y son bastante funcionales para describir qué ocurre en cada sección del código, así, si es necesario modificarlo después, podremos identificar con facilidad sus componentes.

Los comentarios también sirven para “eliminar” líneas de código, pero que podríamos querer utilizar después, por ejemplo, la que dibuja en pantalla cuál fue la última tecla presionada (No queremos que la gente que juegue nuestro juego la vea, ¿O sí?).

Guarda el juego y abre “index.html”. Si todo está de forma correcta, ahora podrás controlar al pequeño rectángulo usando las flechas del teclado. ¡Felicidades!

## Pausa

Podría dar por concluida la lección de hoy, pero aprovechando que estamos viendo como usar el teclado, te contaré un pequeño truco que se usa para poner “Pausa” a un juego. Comenzaremos por supuesto, creando una variable que indicará si el juego está en pausa:

javascript

```

var pause = true;

```

Ahora, encerraremos todo el contenido de nuestra función “act” en una condicional “if (!pause)”, o sea, si el juego no está en pausa. Hasta el momento hemos hecho condicionales de una sola línea, por lo que no hemos usado llaves. Pero en caso de ser utilizada más de una línea en la condicional, se deben usar llaves igual que en las funciones, tal como hacemos en el caso actual.

Al final de la condicional “if (!pause)”, agregaremos estas líneas para que al presionar “KEY\_ENTER” (13), se cambie el juego entre pausado y sin pausa:

javascript

```

// Pause/Unpause
if (lastPress == KEY_ENTER) {
    pause = !pause;
    lastPress = null;
}

```

Es muy importante que estas líneas no las encierres dentro del “if (!pause)”, o de lo contrario, jamás podrás quitar la pausa (Por que jamás entrarás a esa parte del código). La asignación “pause = !pause” indica que cambio su valor por el opuesto (falso si es verdadero o verdadero si es falso), y después nulificamos “lastPress”, o de lo contrario, el juego estaría poniendo y quitando pausa sin fin hasta que se presione otra tecla.

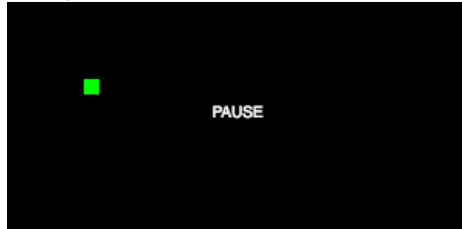
Por último, dibujaremos en nuestra función “paint” el texto “PAUSE” de forma centrada, si la pausa está activada:



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

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

### Código final:



```
1  var KEY_ENTER = 13,
2      KEY_LEFT = 37,
3      KEY_UP = 38,
4      KEY_RIGHT = 39,
5      KEY_DOWN = 40,
6
7      canvas = null,
8      ctx = null,
9      lastPress = null,
10     pause = true,
11     x = 50,
12     y = 50,
13     dir = 0;
14
15 window.requestAnimationFrame = (function () {
16     return window.requestAnimationFrame ||
17         window.mozRequestAnimationFrame ||
18         window.webkitRequestAnimationFrame ||
19         function (callback) {
20             window.setTimeout(callback, 17);
21         };
22 })();
23
24 document.addEventListener('keydown', function (evt) {
25     lastPress = evt.which;
26 }, false);
27
28 function paint(ctx) {
29     // Clean canvas
30     ctx.fillStyle = '#000';
31     ctx.fillRect(0, 0, canvas.width, canvas.height);
32
33     // Draw square
34     ctx.fillStyle = '#0f0';
35     ctx.fillRect(x, y, 10, 10);
36
37     // Debug last key pressed
38     ctx.fillStyle = '#fff';
39     //ctx.fillText('Last Press: ' + lastPress, 0, 20);
40
41     // Draw pause
42     if (pause) {
43         ctx.textAlign = 'center';
44         ctx.fillText('PAUSE', 150, 75);
45         ctx.textAlign = 'left';
46     }
47 }
48
49 function act() {
50     if (!pause) {
51         // Change Direction
52         if (lastPress == KEY_UP) {
```

```

53         dir = 0;
54     }
55     if (lastPress == KEY_RIGHT) {
56         dir = 1;
57     }
58     if (lastPress == KEY_DOWN) {
59         dir = 2;
60     }
61     if (lastPress == KEY_LEFT) {
62         dir = 3;
63     }
64
65     // Move Rect
66     if (dir == 0) {
67         y -= 10;
68     }
69     if (dir == 1) {
70         x += 10;
71     }
72     if (dir == 2) {
73         y += 10;
74     }
75     if (dir == 3) {
76         x -= 10;
77     }
78
79     // Out Screen
80     if (x > canvas.width) {
81         x = 0;
82     }
83     if (y > canvas.height) {
84         y = 0;
85     }
86     if (x < 0) {
87         x = canvas.width;
88     }
89     if (y < 0) {
90         y = canvas.height;
91     }
92 }
93
94 // Pause/Unpause
95 if (lastPress == KEY_ENTER) {
96     pause = !pause;
97     lastPress = null;
98 }
99 }
100
101 function repaint() {
102     window.requestAnimationFrame(repaint);
103     paint(ctx);
104 }
105
106 function run() {
107     setTimeout(run, 50);
108     act();
109 }
110
111 function init() {
112     // Get canvas and context
113     canvas = document.getElementById('canvas');
114     ctx = canvas.getContext('2d');
115
116     // Start game
117     run();
118     repaint();
119 }
120
121 window.addEventListener('load', init, false);

```

## 01.04. Snake - Interactuando con otros elementos.

Además de poder interactuar nosotros con el juego, el segundo punto importante de un juego es que los elementos puedan interactuar entre sí. Para saber si dos elementos “se están tocando” (Es decir, hay una intersección entre ellos), no solo nos basta saber su posición XY, también necesitamos conocer el alto y ancho de los elementos.

En Javascript, las funciones cumplen una doble función. No solo te permiten crear eventos que pueden ser llamados en cualquier momento (como ya hemos visto), si no que además pueden hacer la función de objetos (Esto en otros lenguajes suele ser conocido como “clases”). Por ejemplo, crearemos nuestra propio objeto “rectángulo”, que contendrá una posición en X, en Y, además de un ancho y alto.

Además, **las funciones de tipo objeto pueden contener sus propias funciones**. Así pues, agregaremos al rectángulo una función “intersección”, que nos dirá si está en una intersección con un segundo elemento, así como una función que rellene de forma más sencilla el rectángulo. Para conocer más a detalle sobre este tema, lee el [Apéndice 3: Programación Orientada a Objetos](#); por ahora, tan solo copiemos al final de nuestro código, la siguiente función:

```
function Rectangle(x, y, width, height) {
  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.intersects = function (rect) {
    if (rect == null) {
      window.console.warn('Missing parameters on function intersects');
    } else {
      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);
    }
  };

  this.fill = function (ctx) {
    if (ctx == null) {
      window.console.warn('Missing parameters on function fill');
    } else {
      ctx.fillRect(this.x, this.y, this.width, this.height);
    }
  };
}
```

javascript

Como podremos ver, la función Rectángulo está diseñada para recibir las cuatro variables en el orden X, Y, Ancho y Alto. Si omitimos enviar alguna, esta automáticamente se volverá “0” para evitar errores, excepto en el caso de la Altura, que automáticamente obtendrá el valor del Ancho (Lo que es práctico para omitir enviar ancho y alto si queremos que ambos tengan el mismo valor).

Ahora haremos unos cambios a nuestro código. Primero eliminaremos las variables “x” y “y” que declaramos en un comienzo, y en su lugar, crearemos una variable “player” nula:

```
var player = null;
```

javascript

Al cual asignaremos un valor de nuestro nuevo tipo rectángulo dentro de la función “init”:

```
// Create player
player = new Rectangle(40, 40, 10, 10);
```

javascript

Después, cambiaríamos la forma en que se dibuja el rectángulo a la siguiente forma:

```
player.fill(ctx);
```

javascript

Por último, y hay que tener mucho cuidado en esto, será que todas nuestras variables “x” y “y” que usamos antes las convertiremos agregando “player.” antes de ellas, de igual forma que hicimos con la forma en que dibujamos el rectángulo.

Ahora, necesitaremos un nuevo elemento con el cual interactuar. Crearemos una nueva variable al que asignaremos un valor tipo rectángulo llamada “food”:

```
food = new Rectangle(80, 80, 10, 10);
```

javascript

De igual forma, la dibujaremos, solo que esta la haremos de color rojo:

```
// Draw food
ctx.fillStyle = '#f00';
```

javascript

```
food.fill(ctx);
```

Ahora, analizaremos si ambos están en una intersección. En dado caso, agregaremos un punto a nuestro puntaje, y cambiaremos la posición de la comida a otro lugar al azar. Para ello, primero tendremos que declarar nuestro puntaje:

```
var score = 0;
```

javascript

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

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

javascript

Ahora si, en la función "act", después de mover a nuestro jugador, compararemos si ambos elementos están en una intersección, y de ser así, agregaremos un punto más y cambiaremos de posición la comida:

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

javascript

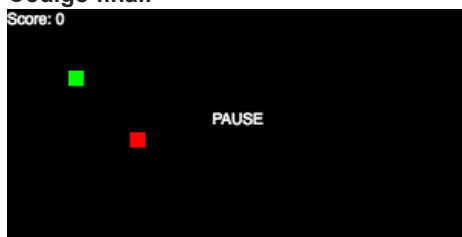
La pequeña ecuación de dividir la pantalla entre 10 dentro del random y multiplicarla al final de nuevo, hace que la comida aparezca en un lugar cada 10 pixeles, de esta forma se ajustará "a la rejilla". Por último, dibujaremos nuestro puntaje en pantalla:

```
// Draw score  
ctx.fillText('Score: ' + score, 0, 10);
```

javascript

Guardemos y probemos el código. Ahora cada vez que el rectángulo verde toque al rojo, el puntaje subirá.

### Código final:



```
1  var KEY_ENTER = 13,  
2    KEY_LEFT = 37,  
3    KEY_UP = 38,  
4    KEY_RIGHT = 39,  
5    KEY_DOWN = 40,  
6  
7    canvas = null,  
8    ctx = null,  
9    lastPress = null,  
10   pause = true,  
11   dir = 0,  
12   score = 0,  
13   player = null,  
14   food = null;  
15  
16   window.requestAnimationFrame = (function () {  
17       return window.requestAnimationFrame ||  
18         window.mozRequestAnimationFrame ||  
19         window.webkitRequestAnimationFrame ||  
20         function (callback) {  
21             window.setTimeout(callback, 17);  
22         };  
23   })();  
24  
25   document.addEventListener('keydown', function (evt) {  
26       lastPress = evt.which;  
27   }, false);  
28  
29   function Rectangle(x, y, width, height) {  
30       this.x = (x == null) ? 0 : x;
```

javascript

```

31  this.y = (y == null) ? 0 : y;
32  this.width = (width == null) ? 0 : width;
33  this.height = (height == null) ? this.width : height;
34
35  this.intersects = function (rect) {
36      if (rect == null) {
37          window.console.warn('Missing parameters on function intersects');
38      } else {
39          return (this.x < rect.x + rect.width &&
40                  this.x + this.width > rect.x &&
41                  this.y < rect.y + rect.height &&
42                  this.y + this.height > rect.y);
43      }
44  };
45
46  this.fill = function (ctx) {
47      if (ctx == null) {
48          window.console.warn('Missing parameters on function fill');
49      } else {
50          ctx.fillRect(this.x, this.y, this.width, this.height);
51      }
52  };
53  }
54
55  function random(max) {
56      return Math.floor(Math.random() * max);
57  }
58
59  function paint(ctx) {
60      // Clean canvas
61      ctx.fillStyle = '#000';
62      ctx.fillRect(0, 0, canvas.width, canvas.height);
63
64      // Draw player
65      ctx.fillStyle = '#0f0';
66      player.fill(ctx);
67
68      // Draw food
69      ctx.fillStyle = '#f00';
70      food.fill(ctx);
71
72      // Debug last key pressed
73      ctx.fillStyle = '#fff';
74      //ctx.fillText('Last Press: '+lastPress,0,20);
75
76      // Draw score
77      ctx.fillText('Score: ' + score, 0, 10);
78
79      // Draw pause
80      if (pause) {
81          ctx.textAlign = 'center';
82          ctx.fillText('PAUSE', 150, 75);
83          ctx.textAlign = 'left';
84      }
85  }
86
87  function act() {
88      if (!pause) {
89          // Change Direction
90          if (lastPress == KEY_UP) {
91              dir = 0;
92          }
93          if (lastPress == KEY_RIGHT) {
94              dir = 1;
95          }
96          if (lastPress == KEY_DOWN) {
97              dir = 2;
98          }
99          if (lastPress == KEY_LEFT) {
100              dir = 3;
101          }
102

```

```

103     // Move Rect
104     if (dir == 0) {
105         player.y -= 10;
106     }
107     if (dir == 1) {
108         player.x += 10;
109     }
110     if (dir == 2) {
111         player.y += 10;
112     }
113     if (dir == 3) {
114         player.x -= 10;
115     }
116
117     // Out Screen
118     if (player.x > canvas.width) {
119         player.x = 0;
120     }
121     if (player.y > canvas.height) {
122         player.y = 0;
123     }
124     if (player.x < 0) {
125         player.x = canvas.width;
126     }
127     if (player.y < 0) {
128         player.y = canvas.height;
129     }
130
131     // Food Intersects
132     if (player.intersects(food)) {
133         score += 1;
134         food.x = random(canvas.width / 10 - 1) * 10;
135         food.y = random(canvas.height / 10 - 1) * 10;
136     }
137 }
138
139 // Pause/Unpause
140 if (lastPress == KEY_ENTER) {
141     pause = !pause;
142     lastPress = null;
143 }
144 }
145
146 function repaint() {
147     window.requestAnimationFrame(repaint);
148     paint(ctx);
149 }
150
151 function run() {
152     setTimeout(run, 50);
153     act();
154 }
155
156 function init() {
157     // Get canvas and context
158     canvas = document.getElementById('canvas');
159     ctx = canvas.getContext('2d');
160
161     // Create player and food
162     player = new Rectangle(40, 40, 10, 10);
163     food = new Rectangle(80, 80, 10, 10);
164
165     // Start game
166     run();
167     repaint();
168 }
169
170 window.addEventListener('load', init, false);

```

## 01.05. Snake - Interactuando con varios elementos iguales.

Ya vimos como hacer para que un objeto interactúe con otro. El problema sería si quisiéramos, por ejemplo, querer interactuar con 50 elementos que hagan exactamente lo mismo (Como serían por ejemplo los enemigos). Tener que evaluar 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 arreglos.

Para este ejemplo, crearemos una variable de tipo arreglo llamada "wall":

```
var wall = new Array();
```

javascript

Este arreglo contendrá todos nuestros elementos de tipo pared. Ahora, agregaremos cuatro elementos a este arreglo en la función "init" de la siguiente forma:

```
// Create walls
wall.push(new Rectangle(100, 50, 10, 10));
wall.push(new Rectangle(100, 100, 10, 10));
wall.push(new Rectangle(200, 50, 10, 10));
wall.push(new Rectangle(200, 100, 10, 10));
```

javascript

Para dibujar los elementos de la pared, recorreremos los elementos del arreglo a través de un "for", de la siguiente forma:

```
// Draw walls
ctx.fillStyle = '#999';
for (i = 0, l = wall.length; i < l; i += 1) {
    wall[i].fill(ctx);
}
```

javascript

De igual forma, comprobaremos cada elemento de la pared con un "for", y comprobaremos si hace una intersección con la comida o el jugador:

```
// Wall Intersects
for (i = 0, l = wall.length; i < l; i += 1) {
    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])) {
        pause = true;
    }
}
```

javascript

Primero, comprobamos si la comida choca con la pared. En dado caso, cambiamos de lugar la comida, esto evitará que esta quede "atorada" en la pared. Segundo, comprobamos si el jugador choca con la pared, y en tal caso, el juego se detendrá. Eso está bien, pero lo ideal sería que cuando el jugador choque, al reanudar el juego, este comience desde el principio.

Arreglemos esto. Comencemos por saber cuando el jugador ha perdido, a través de una variable llamada "gameover":

```
var gameover = true;
```

javascript

Luego, agreguemos estas líneas justo donde el juego comienza, después del "if (!pause)":

```
// GameOver Reset
if (gameover) {
    reset();
}
```

javascript

De esta forma, llamaremos a una función llamada "reset", donde indicaremos como queremos que inicie el juego. En este caso, pondremos el score en cero, la dirección hacia su punto original, regresaremos al jugador a su punto inicial y cambiaremos de lugar la comida. Por último, por supuesto, nos aseguraremos que el juego deje de estar en Game Over:

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

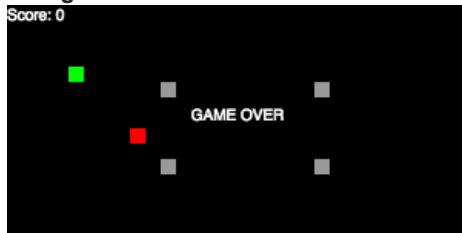
javascript

Por último, cambiaremos el "if (pause)" en nuestra función "paint" para ver si el juego está en Game Over o en una pausa común:

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

De esta forma, concluimos con todos los conocimientos básicos para crear un juego. En el último capítulo, nos enfocaremos en los detalles para darle forma a este juego.

### Código final:



```
1  var KEY_ENTER = 13,
2      KEY_LEFT = 37,
3      KEY_UP = 38,
4      KEY_RIGHT = 39,
5      KEY_DOWN = 40,
6
7      canvas = null,
8      ctx = null,
9      lastPress = null,
10     pause = true,
11     gameover = true,
12     dir = 0,
13     score = 0,
14     wall = new Array(),
15     player = null,
16     food = null;
17
18 window.requestAnimationFrame = (function () {
19     return window.requestAnimationFrame ||
20         window.mozRequestAnimationFrame ||
21         window.webkitRequestAnimationFrame ||
22         function (callback) {
23             window.setTimeout(callback, 17);
24         };
25 })();
26
27 document.addEventListener('keydown', function (evt) {
28     lastPress = evt.which;
29 }, false);
30
31 function Rectangle(x, y, width, height) {
32     this.x = (x == null) ? 0 : x;
33     this.y = (y == null) ? 0 : y;
34     this.width = (width == null) ? 0 : width;
35     this.height = (height == null) ? this.width : height;
36
37     this.intersects = function (rect) {
38         if (rect == null) {
39             window.console.warn('Missing parameters on function intersects');
40         } else {
41             return (this.x < rect.x + rect.width &&
42                 this.x + this.width > rect.x &&
43                 this.y < rect.y + rect.height &&
44                 this.y + this.height > rect.y);
45         }
46     };
47 }
```



```

48     this.fill = function (ctx) {
49         if (ctx == null) {
50             window.console.warn('Missing parameters on function fill');
51         } else {
52             ctx.fillRect(this.x, this.y, this.width, this.height);
53         }
54     };
55 }
56
57 function random(max) {
58     return Math.floor(Math.random() * max);
59 }
60
61 function reset() {
62     score = 0;
63     dir = 1;
64     player.x = 40;
65     player.y = 40;
66     food.x = random(canvas.width / 10 - 1) * 10;
67     food.y = random(canvas.height / 10 - 1) * 10;
68     gameover = false;
69 }
70
71 function paint(ctx) {
72     var i = 0,
73         l = 0;
74
75     // Clean canvas
76     ctx.fillStyle = '#000';
77     ctx.fillRect(0, 0, canvas.width, canvas.height);
78
79     // Draw player
80     ctx.fillStyle = '#0f0';
81     player.fill(ctx);
82
83     // Draw walls
84     ctx.fillStyle = '#999';
85     for (i = 0, l = wall.length; i < l; i += 1) {
86         wall[i].fill(ctx);
87     }
88
89     // Draw food
90     ctx.fillStyle = '#f00';
91     food.fill(ctx);
92
93     // Debug last key pressed
94     ctx.fillStyle = '#fff';
95     //ctx.fillText('Last Press: '+lastPress,0,20);
96
97     // Draw score
98     ctx.fillText('Score: ' + score, 0, 10);
99
100    // Draw pause
101    if (pause) {
102        ctx.textAlign = 'center';
103        if (gameover) {
104            ctx.fillText('GAME OVER', 150, 75);
105        } else {
106            ctx.fillText('PAUSE', 150, 75);
107        }
108        ctx.textAlign = 'left';
109    }
110 }
111
112 function act() {
113     var i,
114         l;
115
116     if (!pause) {
117         // GameOver Reset
118         if (gameover) {
119             reset();

```

```

120     }
121
122     // Change Direction
123     if (lastPress == KEY_UP) {
124         dir = 0;
125     }
126     if (lastPress == KEY_RIGHT) {
127         dir = 1;
128     }
129     if (lastPress == KEY_DOWN) {
130         dir = 2;
131     }
132     if (lastPress == KEY_LEFT) {
133         dir = 3;
134     }
135
136     // Move Rect
137     if (dir == 0) {
138         player.y -= 10;
139     }
140     if (dir == 1) {
141         player.x += 10;
142     }
143     if (dir == 2) {
144         player.y += 10;
145     }
146     if (dir == 3) {
147         player.x -= 10;
148     }
149
150     // Out Screen
151     if (player.x > canvas.width) {
152         player.x = 0;
153     }
154     if (player.y > canvas.height) {
155         player.y = 0;
156     }
157     if (player.x < 0) {
158         player.x = canvas.width;
159     }
160     if (player.y < 0) {
161         player.y = canvas.height;
162     }
163
164     // Food Intersects
165     if (player.intersects(food)) {
166         score += 1;
167         food.x = random(canvas.width / 10 - 1) * 10;
168         food.y = random(canvas.height / 10 - 1) * 10;
169     }
170
171     // Wall Intersects
172     for (i = 0, l = wall.length; i < l; i += 1) {
173         if (food.intersects(wall[i])) {
174             food.x = random(canvas.width / 10 - 1) * 10;
175             food.y = random(canvas.height / 10 - 1) * 10;
176         }
177
178         if (player.intersects(wall[i])) {
179             gameover = true;
180             pause = true;
181         }
182     }
183 }
184
185 // Pause/Unpause
186 if (lastPress == KEY_ENTER) {
187     pause = !pause;
188     lastPress = null;
189 }
190 }
191

```

```
192 function repaint() {
193     window.requestAnimationFrame(repaint);
194     paint(ctx);
195 }
196
197 function run() {
198     setTimeout(run, 50);
199     act();
200 }
201
202 function init() {
203     // Get canvas and context
204     canvas = document.getElementById('canvas');
205     ctx = canvas.getContext('2d');
206
207     // Create player and food
208     player = new Rectangle(40, 40, 10, 10);
209     food = new Rectangle(80, 80, 10, 10);
210
211     // Create walls
212     wall.push(new Rectangle(100, 50, 10, 10));
213     wall.push(new Rectangle(100, 100, 10, 10));
214     wall.push(new Rectangle(200, 50, 10, 10));
215     wall.push(new Rectangle(200, 100, 10, 10));
216
217     // Start game
218     run();
219     repaint();
220 }
221
222 window.addEventListener('load', init, false);
```

## 01.06. Snake - 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.

Para empezar, comentaremos todo el código con respecto a nuestras paredes para que estas no estorben en el desarrollo y prueba de nuestro juego. Posteriormente será decisión de cada uno si incluirlas o no, y de que forma, para que cada juego sea único.

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

```
//player = null,
body = new Array(),
```

javascript

Y cada llamada a “player”, será sustituida por “body[0]”, que es la cabeza de la serpiente. Así pues, donde se movía “player.x” y “player.y”, ahora se moverá “body[0].x” y “body[0].y”. Podemos utilizar la función “reemplazar” de nuestro editor de texto (normalmente Editar » Buscar y Reemplazar) para hacer estas tareas de forma automática.

También es importante resaltar en la función “paint”, que ya no es solo la cabeza la que dibujaremos, si no todo el cuerpo, y esto haremos mediante un “for”:

```
// Draw player
ctx.fillStyle = '#0f0';
for (i = 0, l = body.length; i < l; i += 1) {
  body[i].fill(ctx);
}
```

javascript

Para que nuestra serpiente tenga siempre la misma longitud al comienzo de cada juego, debemos “cortarla” a cero, y luego agregar cada parte de su cuerpo tanto larga como la deseemos. Esto se hace en la función “reset” de esta forma:

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

javascript

Es importante que la primer parte del cuerpo (la cabeza) esté en la posición que deseemos que inicie. El resto le seguirá después.

Para mover el cuerpo de la serpiente, se hace uso de un truco peculiar. Este se debe mover de atrás hacia adelante, y antes de moverse la cabeza, haciendo así un efecto de oruga, en que la cola va “empujando” el resto del cuerpo, mediante el siguiente “for”:

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

javascript

De hacerlo de la forma opuesta, todo el cuerpo estaría siempre ocupando el mismo sitio, y no solo “no se vería nada”, si no que además, eso provocaría la perdida del juego al chocar la cabeza con su cuerpo. Para comprobar si el cuerpo choca con la cabeza, se hará de igual forma que como comprobábamos si el personaje chocaba con la pared:

```
// Body Intersects
for (i = 2, l = body.length; i < l; i += 1) {
  if (body[0].intersects(body[i])) {
    gameover = true;
    pause = true;
  }
}
```

javascript

Nótese que empezamos a contar desde la segunda parte del cuerpo. Esto es por que la parte 0 es la cabeza, y la 1, el cuello. Si empezáramos desde uno de estos, es posible que el juego quedara en constante Game Over “sin razón aparente”.

Por último y muy importante, es hacer crecer a la serpiente. Esto se hace (lógicamente) cuando la cabeza choca contra la manzana, agregando esta línea justo antes de aumentar el score:

```
body.push(new Rectangle(food.x, food.y, 10, 10));
```

javascript

Guardemos y actualicemos la página. Con esto, nuestro juego queda concluido, y tendremos un clásico juego de la serpiente, sencillo y completo.

### Media

Antes de dar por concluido este pequeño curso, veremos como incluir medios externos en nuestro juego, me refiero claro a imágenes y sonido,

algo muy deseado en el desarrollo de los juegos.

Si no he trabajado con imágenes desde un comienzo, es por el importante hecho de resaltar que el tamaño de las imágenes es independiente al tamaño de nuestros rectángulos en nuestro juego.

Actualmente nuestros rectángulos son de 10x10 píxeles, y sin incluimos imágenes más chicas o más grandes, podría aparentar que los objetos en la pantalla no se tocan o viceversa, cuando en realidad ocurriría lo contrario. Es por eso que es sumamente importante que las imágenes tengan siempre el tamaño de nuestros rectángulos en el juego (excepto claro, cuando se usan técnicas avanzadas de efectos visuales).

Para empezar, abramos nuestro editor de imágenes favoritos (Puede ser MS Paint o cualquiera que tengamos), y en un área de 10x10 píxeles, dibujaremos dos dibujos: nuestra comida (una fruta), y la parte del cuerpo de nuestra serpiente (Yo le hice un círculo con manchas, pero pueden dibujarle como les guste).

Estas dos imágenes ("body.png" y "fruit.png") se guardarán en una carpeta llamada "assets", dentro de la misma carpeta que nuestro código (es sumamente importante que siempre esté la carpeta "assets" junto al código, o las imágenes no se verán). Ahora, crearemos nuestras variables de imagen:

```
var iBody = new Image(),  
    iFood = new Image();
```

javascript

Y les asignaremos la ruta a la fuente en la función "init":

```
// Load assets  
iBody.src = 'assets/body.png';  
iFood.src = 'assets/fruit.png';
```

javascript

Ahora modifiquemos la función "paint", comentando donde dibujamos los rectángulos del cuerpo y la comida, y dibujando las imágenes que hemos hecho en su lugar:

```
// Draw player  
//ctx.fillStyle = '#0f0';  
for (i = 0, l = body.length; i < l; i += 1) {  
    //body[i].fill(ctx);  
    ctx.drawImage(iBody, body[i].x, body[i].y);  
}  
  
// Draw food  
//ctx.fillStyle = '#f00';  
//food.fill(ctx);  
ctx.drawImage(iFood, food.x, food.y);
```

javascript

Al guardar y actualizar la página, veremos que ahora 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. Puedes conseguir sonidos para tu juego buscándoles en Internet. La declaración será de esta forma:

```
var aEat = new Audio(),  
    aDie = new Audio();
```

javascript

Y se le asigna estos valores en la función "init":

```
aEat.src = 'assets/chomp.oga';  
aDie.src = 'assets/dies.oga';
```

javascript

Para reproducirlos, los agregamos en las área de colisión correspondiente (con la manzana y con el cuerpo), reproduciéndoles así:

```
aEat.play();  
aDie.play();
```

javascript

Con esto concluye este pequeño curso, el cual espero haya sido de ayuda y agrada para ustedes. Si tienen dudas, estén seguros que responderé sus comentarios. También pueden enviarme enlaces a los juegos que creen gracias a este curso, los cuales estaré encantado de conocer.

¡Felices códigos!

## Recursos:

(Usa clic derecho y "Guardar vínculo como" para guardar estos archivos).

- [body.png](#)
- [fruit.png](#)
- [chomp.oga](#)
- [dies.oga](#)

## Código Final:

javascript

```
1  var KEY_ENTER = 13,
2      KEY_LEFT = 37,
3      KEY_UP = 38,
4      KEY_RIGHT = 39,
5      KEY_DOWN = 40,
6
7      canvas = null,
8      ctx = null,
9      lastPress = null,
10     pause = true,
11     gameover = true,
12     dir = 0,
13     score = 0,
14     //wall = new Array(),
15     body = new Array(),
16     food = null,
17     iBody = new Image(),
18     iFood = new Image(),
19     aEat = new Audio(),
20     aDie = new Audio();
21
22 window.requestAnimationFrame = (function () {
23     return window.requestAnimationFrame ||
24         window.mozRequestAnimationFrame ||
25         window.webkitRequestAnimationFrame ||
26         function (callback) {
27             window.setTimeout(callback, 17);
28         };
29 })();
30
31 document.addEventListener('keydown', function (evt) {
32     lastPress = evt.which;
33 }, false);
34
35 function Rectangle(x, y, width, height) {
36     this.x = (x == null) ? 0 : x;
37     this.y = (y == null) ? 0 : y;
38     this.width = (width == null) ? 0 : width;
39     this.height = (height == null) ? this.width : height;
40
41     this.intersects = function (rect) {
42         if (rect == null) {
43             window.console.warn('Missing parameters on function intersects');
44         } else {
45             return (this.x < rect.x + rect.width &&
46                 this.x + this.width > rect.x &&
47                 this.y < rect.y + rect.height &&
48                 this.y + this.height > rect.y);
49         }
50     };
51
52     this.fill = function (ctx) {
53         if (ctx == null) {
54             window.console.warn('Missing parameters on function fill');
55         } else {
56             ctx.fillRect(this.x, this.y, this.width, this.height);
57         }
58     };
59 }
60
```

```

61 function random(max) {
62     return Math.floor(Math.random() * max);
63 }
64
65 function reset() {
66     score = 0;
67     dir = 1;
68     body.length = 0;
69     body.push(new Rectangle(40, 40, 10, 10));
70     body.push(new Rectangle(0, 0, 10, 10));
71     body.push(new Rectangle(0, 0, 10, 10));
72     food.x = random(canvas.width / 10 - 1) * 10;
73     food.y = random(canvas.height / 10 - 1) * 10;
74     gameover = false;
75 }
76
77 function paint(ctx) {
78     var i = 0,
79         l = 0;
80
81     // Clean canvas
82     ctx.fillStyle = '#000';
83     ctx.fillRect(0, 0, canvas.width, canvas.height);
84
85     // Draw player
86     //ctx.fillStyle = '#0f0';
87     for (i = 0, l = body.length; i < l; i += 1) {
88         //body[i].fill(ctx);
89         ctx.drawImage(iBody, body[i].x, body[i].y);
90     }
91
92     // Draw walls
93     //ctx.fillStyle = '#999';
94     //for(i = 0 ,l = wall.length; i < l; i += 1) {
95     //    wall[i].fill(ctx);
96     //}
97
98     // Draw food
99     //ctx.fillStyle = '#f00';
100    //food.fill(ctx);
101    ctx.drawImage(iFood, food.x, food.y);
102
103    // Debug last key pressed
104    ctx.fillStyle = '#fff';
105    //ctx.fillText('Last Press: ' + lastPress, 0, 20);
106
107    // Draw score
108    ctx.fillText('Score: ' + score, 0, 10);
109
110    // Draw pause
111    if (pause) {
112        ctx.textAlign = 'center';
113        if (gameover) {
114            ctx.fillText('GAME OVER', 150, 75);
115        } else {
116            ctx.fillText('PAUSE', 150, 75);
117        }
118        ctx.textAlign = 'left';
119    }
120 }
121
122 function act() {
123     var i = 0,
124         l = 0;
125
126     if (!pause) {
127         // GameOver Reset
128         if (gameover) {
129             reset();
130         }
131
132         // Move Body

```

```

133     for (i = body.length - 1; i > 0; i -= 1) {
134         body[i].x = body[i - 1].x;
135         body[i].y = body[i - 1].y;
136     }
137
138     // Change Direction
139     if (lastPress == KEY_UP && dir != 2) {
140         dir = 0;
141     }
142     if (lastPress == KEY_RIGHT && dir != 3) {
143         dir = 1;
144     }
145     if (lastPress == KEY_DOWN && dir != 0) {
146         dir = 2;
147     }
148     if (lastPress == KEY_LEFT && dir != 1) {
149         dir = 3;
150     }
151
152     // Move Head
153     if (dir == 0) {
154         body[0].y -= 10;
155     }
156     if (dir == 1) {
157         body[0].x += 10;
158     }
159     if (dir == 2) {
160         body[0].y += 10;
161     }
162     if (dir == 3) {
163         body[0].x -= 10;
164     }
165
166     // Out Screen
167     if (body[0].x > canvas.width - body[0].width) {
168         body[0].x = 0;
169     }
170     if (body[0].y > canvas.height - body[0].height) {
171         body[0].y = 0;
172     }
173     if (body[0].x < 0) {
174         body[0].x = canvas.width - body[0].width;
175     }
176     if (body[0].y < 0) {
177         body[0].y = canvas.height - body[0].height;
178     }
179
180     // Wall Intersects
181     //for(i = 0, l = wall.length; i < l; i += 1){
182     //    if (food.intersects(wall[i])) {
183     //        food.x = random(canvas.width / 10 - 1) * 10;
184     //        food.y = random(canvas.height / 10 - 1) * 10;
185     //    }
186     //
187     //    if(body[0].intersects(wall[i])){
188     //        gameover = true;
189     //        pause = true;
190     //    }
191     //}
192
193     // Body Intersects
194     for (i = 2, l = body.length; i < l; i += 1) {
195         if (body[0].intersects(body[i])) {
196             gameover = true;
197             pause = true;
198             aDie.play();
199         }
200     }
201
202     // Food Intersects
203     if (body[0].intersects(food)) {
204         body.push(new Rectangle(food.x, food.y, 10, 10));

```



```

205         score += 1;
206         food.x = random(canvas.width / 10 - 1) * 10;
207         food.y = random(canvas.height / 10 - 1) * 10;
208         aEat.play();
209     }
210 }
211
212 // Pause/Unpause
213 if (lastPress == KEY_ENTER) {
214     pause = !pause;
215     lastPress = null;
216 }
217 }
218
219 function repaint() {
220     window.requestAnimationFrame(repaint);
221     paint(ctx);
222 }
223
224 function run() {
225     setTimeout(run, 50);
226     act();
227 }
228
229 function init() {
230     // Get canvas and context
231     canvas = document.getElementById('canvas');
232     ctx = canvas.getContext('2d');
233
234     // Load assets
235     iBody.src = 'assets/body.png';
236     iFood.src = 'assets/fruit.png';
237     aEat.src = 'assets/chomp.oga';
238     aDie.src = 'assets/dies.oga';
239
240     // Create food
241     food = new Rectangle(80, 80, 10, 10);
242
243     // Create walls
244     //wall.push(new Rectangle(100, 50, 10, 10));
245     //wall.push(new Rectangle(100, 100, 10, 10));
246     //wall.push(new Rectangle(200, 50, 10, 10));
247     //wall.push(new Rectangle(200, 100, 10, 10));
248
249     // Start game
250     run();
251     repaint();
252 }
253
254 window.addEventListener('load', init, false);

```

## 01.07. Snake - Optimización para Javascript

Aunque el presente curso ha sido enseñado en el lenguaje Javascript, sus conocimientos son globales, por lo que adaptar lo aquí aprendido a un nuevo lenguaje, sería relativamente sencillo.

Sin embargo, si deseas proseguir tu camino por el camino del desarrollo de juegos para la web, es recomendable seguir las recomendaciones siguientes, para optimizar tus juegos en este lenguaje:

### Encapsular el código en una función auto-ejecutable

Una de las características peculiares en Javascript, es que todos los scripts son globales y pueden interactuar entre sí. A veces eso puede ser una ventaja, pero en consecuencia, puede llevar a efectos no deseados, como la redefinición de una variable con el mismo nombre.

Por ejemplo, si tu código usa una variable "x" para almacenar la posición de un objeto en tu juego, pero otro script usa otra variable con el mismo nombre "x" para una acción diferente, la variable eventualmente tendrá un valor distinto al planeado, y esto creará conflictos para uno de los dos códigos, si no ambos.

Para evitar esto, es importante encapsular los códigos de Javascript en una función auto-ejecutable, poniendo todo su contenido dentro de un código como el del ejemplo a continuación:

```
(function (window, undefined) {  
    //...  
})(window);
```

javascript

De esta forma, todas las variables y objetos dentro de nuestro código, serán exclusivamente locales, y no entrarán en conflicto con cualquier otra variable que quede de forma global. Nuestro código aun podrá usar variables globales si es necesario, pero los scripts externos no podrán modificar las variables dentro de nuestro código.

Notarás que la función recibe las variables "window" y "undefined", y envía al momento de construirse la variable "window". Aunque esto no parezca tener mucho sentido, es una medida de seguridad de Javascript que previene que funciones dentro de "window" sean reemplazadas posteriormente durante su ejecución, y asegura que "undefined" sea realmente un valor indefinido y no sea reemplazado por otros scripts.

### Usar el modo estricto

Javascript ha evolucionado con el tiempo desde un lenguaje muy básico y permisible para la web, pero aun cuando va mejorando con el tiempo, es necesario que sea compatible con prácticas antiguas que han caído en desuso, y en ocasiones, puede ser incluso contraproducente para el desarrollo en aplicaciones modernas.

Los navegadores no tienen forma de saber si nosotros estamos usando un código con estas antiguas prácticas o no, pero nosotros podemos indicarle que estamos creando un código actual que no depende de estas, agregando al comienzo la siguiente línea:

```
'use strict';
```

javascript

Al usar la versión estricta de Javascript, el navegador desactivará el funcionamiento de todas estas malas prácticas que se hacían antiguamente. Si por error, llegáramos a incluir una, esta será reportada en la Consola de Javascript como un error.

Recuerda que si tu código solo muestra una pantalla gris, debes verificar la Consola de Javascript, tal como aprendimos desde la lección 1.

### Verificar si la imagen está cargada

A veces las imágenes grandes pueden tardar en cargar un momento, o hay ocasiones en las que por un error, la imagen simplemente no carga en lo absoluto. Sin embargo, aun sin una imagen, el juego será ejecutado, confrontando obstáculos invisibles.

La mejor forma de saber si una imagen ha cargado correctamente en Javascript, es verificando su ancho. Si la imagen no ha cargado, esta regresará un valor de 0 o nulo.

Si la imagen no ha sido cargada, es una buena idea poner algo en su lugar que indique que hay una imagen sin dibujarse en ese lugar; de esta forma, el usuario sabrá que ahí está alguna especie de obstáculo, aunque no sepa precisamente de que clase de obstáculo se trate.

Para facilitar esta tarea, se puede agregar este método a la función "Rectangle", que dibuja la imagen en la posición indicada, y si la imagen no ha cargado, dibujará el contorno del rectángulo en su lugar:

```
this.drawImage = function (ctx, img) {  
    if (img == null) {  
        window.console.warn('Missing parameters on function drawImage');  
    } else {  
        if (img.width) {  
            ctx.drawImage(img, this.x, this.y);  
        } else {  
            ctx.strokeRect(this.x, this.y, this.width, this.height);  
        }  
    }  
}
```

javascript

```
    }  
};
```

He preferido usar "strokeRect" en lugar de "fillRect", ya que envía con mayor certeza esa sensación de "Aquí va algo que falta". Puedes combinarlo con los "fillStyle" comentados en el código, cambiándolos por "strokeStyle", creando así un juego bastante funcional, aun en el peor de los casos, en que una imagen falle en cargar (O tarde demasiado en hacerlo).

Para dibujar las imágenes en la función "paint" con esta función personalizada, debes cambiarlas del formato:

```
ctx.drawImage(iFood, food.x, food.y);
```

javascript

A la siguiente forma:

```
food.drawImage(ctx, iFood);
```

javascript

## Compatibilidad de audio

Sobre los sonidos, hay que resaltar un problema importante, pues actualmente, los navegadores Internet Explorer y Safari solo admiten formato mp3 y mp4 (m4a/aac), mientras que Firefox y Opera solo admiten formato ogg (oga). Chrome admite ambos formatos, así que no hay problema con él.

Por tanto, al agregar sonidos en nuestro juego, es posible que en algunos navegadores no se escuche (dependiendo el formato seleccionado).

Hay soluciones avanzadas que permiten descubrir las funciones del navegador del usuario y usar un archivo diferente dependiendo las mismas. La función a continuación nos da a conocer si debe ser usado un archivo de audio ogg o de formato diferente:

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

javascript

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

```
if (canPlayOgg()) {  
    aEat.src="assets/chomp.oga";  
} else {  
    aEat.src="assets/chomp.m4a";  
}
```

javascript

## Usar [] en lugar de "new Array()"

Una forma alterna de crear un nuevo arreglo en Javascript, es simplemente declararlo con un par de corchetes, de la siguiente forma:

```
var body = [];
```

javascript

No solo es mas rápido de escribir y recordar, también resulta más veloz en tiempo de ejecución, por lo que se recomienda su uso en lugar de la forma tradicional.

## Usar ~~ en lugar de "Math.floor()"

De forma similar que con los corchetes en lugar del nuevo arreglo, esta es una forma más corta y veloz de convertir un número decimal a un número entero. Por ejemplo, la función "random" sería modificada a la forma siguiente:

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

javascript

En realidad esta no es una forma alterna de llamar a "Math.floor", si no un truco que da la casualidad de ejecutar la tarea requerida en Javascript, en una forma más eficiente (Y también, más fácil de recordar).

Esta técnica forma parte de un conjunto de técnicas llamada "Operaciones a nivel de bits" (En ingles "Bitwise operation"). Estas operaciones manipulan los bits de un valor, es decir, los números binarios que lo conforman al nivel más básico del lenguaje computadora.

En realidad esto no importa mucho a nuestro conocimiento por ahora, pero diré en forma resumida, que el operador "~", hace una negación de los bits en un valor, es decir, convierte todos los 0s en 1s, y los 1s en 0s. Las operaciones a nivel de bits en Javascript solo pueden usar números enteros, por lo que los decimales son eliminados. Al hacer una doble negación "~~", terminamos con el valor entero de este número decimal, y

como dije antes, en un tiempo de ejecución aun menor al "Math.floor()".

### Usar !== en lugar de != al comparar

Y de igual forma, "===" en lugar de "==". A diferencia de muchos otros lenguajes, Javascript permite hacer comparaciones entre valores equivalentes que no sean estrictamente idénticos, esto permitió versatilidad durante mucho tiempo para el desarrollo web, pero ha causado conflictos cuando se requiere comparar dos valores a que sean exactamente iguales. Para resolver este conflicto, Javascript agregó dos nuevos comparadores para asegurar la estricta igualdad o diferencia de dos valores, agregando un caracter extra de igualdad al final de la comparación.

Para evitar posibles conflictos en el código, es recomendado utilizar siempre los comparadores estrictos al programar en Javascript. Si deseas conocer más sobre la equivalencia de valores en comparadores no estrictos, puedes ver la siguiente tabla: <http://dorey.github.io/JavaScript-Equality-Table/>

```
if (dir === 0) {  
  body[0].y -= 10;  
}
```

javascript

### Comparar con "undefined" en lugar de "null" para valores indefinidos

Al comienzo de este artículo se habló sobre una variable de valor indefinido, pero no se detalló al respecto. En la mayoría de los lenguajes, todos los objetos se inicializan con un valor nulo, y las variables suelen tener el mínimo valor predefinido, sin embargo, en Javascript todas las variables son inicializadas literalmente con un valor "indefinido".

Como se había comentado antes, en Javascript no existía una palabra reservada para representar este valor indefinido, pero por estándar se ha utilizado la variable con valor indefinido "undefined". Cuando se realiza una comparación estricta en Javascript, este es el valor que tienen por defecto las variables sin definir, por ejemplo, al no asignarlas cuando se llama una función; por ello es que se debe hacer esta comparación con "undefined" en lugar de utilizar "null", que sería el valor predeterminado en otros lenguajes.

```
if (img === undefined) {  
  window.console.warn('Missing parameters on function drawImage');  
}
```

javascript

### Usar prototipos en las pseudo-clases.

Se aprendió que JavaScript no tiene clases como tal, a diferencia de otros lenguajes, 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, funcionen de la forma deseada.

Sin embargo, el método aprendido de incluir las funciones en el interior de la pseudo-clase (Que es la forma estándar en que funcionan la mayoría de los lenguajes), no es el mas 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.

JavaScript trabaja realmente en base a prototipos en lugar de clases, y usar esta forma no copia de nuevo la información en la memoria RAM, 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 agregar funciones a un prototipo es diferente a como se hace con clases.

Una forma de hacerlo es crear cada función directamente en el prototipo de la función principal, tal como se muestra a continuación para el prototipo de la función "Rectangle":

```
function Rectangle(x, y, width, height) {  
  this.x = (x === undefined) ? 0 : x;  
  this.y = (y === undefined) ? 0 : y;  
  this.width = (width === undefined) ? 0 : width;  
  this.height = (height === undefined) ? this.width : height;  
}  
  
Rectangle.prototype.intersects = function (rect) {  
  if (rect === undefined) {  
    window.console.warn('Missing parameters on function intersects');  
  } else {  
    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 (ctx) {  
  if (ctx === undefined) {  
    window.console.warn('Missing parameters on function fill');  
  }  
};
```

javascript

```

    } else {
        ctx.fillRect(this.x, this.y, this.width, this.height);
    }
};

Rectangle.prototype.drawImage = function (ctx, img) {
    if (img === undefined) {
        window.console.warn('Missing parameters on function drawImage');
    } else {
        if (img.width) {
            ctx.drawImage(img, this.x, this.y);
        } else {
            ctx.strokeRect(this.x, this.y, this.width, this.height);
        }
    }
};

```

La otra forma es sobrescribir por completo el prototipo con un nuevo objeto que contenga todas las funciones y propiedades deseadas. Si se usa este método hay que reasignar su constructor, o de lo contrario quedará indefinido al sobrescribir dicho prototipo:

javascript

```

function Rectangle(x, y, width, height) {
    this.x = (x === undefined) ? 0 : x;
    this.y = (y === undefined) ? 0 : y;
    this.width = (width === undefined) ? 0 : width;
    this.height = (height === undefined) ? this.width : height;
}

Rectangle.prototype = {
    constructor: Rectangle,

    intersects: function (rect) {
        if (rect === undefined) {
            window.console.warn('Missing parameters on function intersects');
        } else {
            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);
        }
    },

    fill: function (ctx) {
        if (ctx === undefined) {
            window.console.warn('Missing parameters on function fill');
        } else {
            ctx.fillRect(this.x, this.y, this.width, this.height);
        }
    },

    drawImage: function (ctx, img) {
        if (img === undefined) {
            window.console.warn('Missing parameters on function drawImage');
        } else {
            if (img.width) {
                ctx.drawImage(img, this.x, this.y);
            } else {
                ctx.strokeRect(this.x, this.y, this.width, this.height);
            }
        }
    }
};

```

### Seguir los lineamientos de JSLint

JSLint fue creado como una herramienta para código de calidad para Javascript, no solo verifica si el código ha sido escrito correctamente, si no que además muestra una serie de recomendaciones para que el código sea legible y sencillo de comprender, lo cual es de mucha ayuda en proyectos colaborativos, por lo que resulta una buena práctica seguir estos lineamientos desde el comienzo. Herramientas de desarrollo web como [Brackets](#) ya lo incluyen por defecto al revisar código JavaScript.

Hay ciertas recomendaciones activadas por defecto que pueden ser personalizadas agregando comentarios especiales al comienzo del código. Para que nuestro código se muestre válido, tendremos que agregar estas dos líneas al comienzo:

```
/*jslint bitwise:true, es5: true */
```

La primer condición es activar las operaciones a nivel de bits (bitwise), ya que JSLint previene de su uso dado que muy poca gente comprende como funcionan y pueden escribir líneas de código con un propósito distinto al que imaginaban. Nosotros aprendimos poco arriba para que es el comando a nivel de bits que usamos y su razón, por lo que no necesitamos que nos advierta más de los riesgos de usarlo.

La segunda condición es activar "EcmaScript5" (es5). Usualmente JSLint advierte que redefinir "undefined" es posiblemente un error, y eso es cierto, pero en nuestro caso lo hacemos precisamente para prevenir errores por casos donde se llegue a dar esta situación. Esta práctica se hizo popular por seguridad a partir de EcmaScript5 (En el que se basa JavaScript 1.5), por tanto, al especificar que esta es la versión que estamos usando, nos dejará de lanzar esta advertencia, suponiendo que le estamos redefiniendo por la razón correcta.

## Conclusión

Con estos consejos, tus códigos en Javascript se ejecutarán de forma más segura y veloz en futuros proyectos. Como ejercicio, aplica todo lo aquí aprendido al juego que acabamos de desarrollar sin ver el código final. Úsalo solo para compararlo con tu resultado. Si tienes dudas, puedes consultarme en los comentarios.

¡Mucha suerte! ¡Y felices códigos!

```

1  /*jslint bitwise:true, es5: true */
2  (function (window, undefined) {
3      'use strict';
4      var KEY_ENTER = 13,
5          KEY_LEFT = 37,
6          KEY_UP = 38,
7          KEY_RIGHT = 39,
8          KEY_DOWN = 40,
9
10         canvas = null,
11         ctx = null,
12         lastPress = null,
13         pause = true,
14         gameover = true,
15         dir = 0,
16         score = 0,
17         //wall = [],
18         body = [],
19         food = null,
20         iBody = new Image(),
21         iFood = new Image(),
22         aEat = new Audio(),
23         aDie = new Audio();
24
25         window.requestAnimationFrame = (function () {
26             return window.requestAnimationFrame ||
27                 window.mozRequestAnimationFrame ||
28                 window.webkitRequestAnimationFrame ||
29                 function (callback) {
30                     window.setTimeout(callback, 17);
31                 };
32         })();
33
34         document.addEventListener('keydown', function (evt) {
35             lastPress = evt.which;
36         }, false);
37
38         function Rectangle(x, y, width, height) {
39             this.x = (x === undefined) ? 0 : x;
40             this.y = (y === undefined) ? 0 : y;
41             this.width = (width === undefined) ? 0 : width;
42             this.height = (height === undefined) ? this.width : height;
43
44             /*this.intersects = function (rect) {
45                 if (rect === undefined) {
46                     window.console.warn('Missing parameters on function intersects');
47                 } else {
48                     return (this.x < rect.x + rect.width &&
49                         this.x + this.width > rect.x &&
50                         this.y < rect.y + rect.height &&
51                         this.y + this.height > rect.y);
52                 }
44         */

```

```

51     }
52 };
53
54 this.fill = function (ctx) {
55     if (ctx === undefined) {
56         window.console.warn('Missing parameters on function fill');
57     } else {
58         ctx.fillRect(this.x, this.y, this.width, this.height);
59     }
60 };
61
62 this.drawImage = function (ctx, img) {
63     if (img === undefined) {
64         window.console.warn('Missing parameters on function drawImage');
65     } else {
66         if (img.width) {
67             ctx.drawImage(img, this.x, this.y);
68         } else {
69             ctx.strokeRect(this.x, this.y, this.width, this.height);
70         }
71     }
72 };*/
73 }
74
75 Rectangle.prototype = {
76     constructor: Rectangle,
77
78     intersects: function (rect) {
79         if (rect === undefined) {
80             window.console.warn('Missing parameters on function intersects');
81         } else {
82             return (this.x < rect.x + rect.width &&
83                 this.x + this.width > rect.x &&
84                 this.y < rect.y + rect.height &&
85                 this.y + this.height > rect.y);
86         }
87     },
88
89     fill: function (ctx) {
90         if (ctx === undefined) {
91             window.console.warn('Missing parameters on function fill');
92         } else {
93             ctx.fillRect(this.x, this.y, this.width, this.height);
94         }
95     },
96
97     drawImage: function (ctx, img) {
98         if (img === undefined) {
99             window.console.warn('Missing parameters on function drawImage');
100     } else {
101         if (img.width) {
102             ctx.drawImage(img, this.x, this.y);
103         } else {
104             ctx.strokeRect(this.x, this.y, this.width, this.height);
105         }
106     }
107 }
108 };
109
110 /*Rectangle.prototype.intersects = function (rect) {
111     if (rect === undefined) {
112         window.console.warn('Missing parameters on function intersects');
113     } else {
114         return (this.x < rect.x + rect.width &&
115             this.x + this.width > rect.x &&
116             this.y < rect.y + rect.height &&
117             this.y + this.height > rect.y);
118     }
119 };
120
121 Rectangle.prototype.fill = function (ctx) {
122     if (ctx === undefined) {

```

```

123         window.console.warn('Missing parameters on function fill');
124     } else {
125         ctx.fillRect(this.x, this.y, this.width, this.height);
126     }
127 };
128
129 Rectangle.prototype.drawImage = function (ctx, img) {
130     if (img === undefined) {
131         window.console.warn('Missing parameters on function drawImage');
132     } else {
133         if (img.width) {
134             ctx.drawImage(img, this.x, this.y);
135         } else {
136             ctx.strokeRect(this.x, this.y, this.width, this.height);
137         }
138     }
139 };*/
140
141 function random(max) {
142     return ~~(Math.random() * max);
143 }
144
145 function canPlayOgg() {
146     var aud = new Audio();
147     if (aud.canPlayType('audio/ogg').replace(/no/, '')) {
148         return true;
149     } else {
150         return false;
151     }
152 }
153
154 function reset() {
155     score = 0;
156     dir = 1;
157     body.length = 0;
158     body.push(new Rectangle(40, 40, 10, 10));
159     body.push(new Rectangle(0, 0, 10, 10));
160     body.push(new Rectangle(0, 0, 10, 10));
161     food.x = random(canvas.width / 10 - 1) * 10;
162     food.y = random(canvas.height / 10 - 1) * 10;
163     gameOver = false;
164 }
165
166 function paint(ctx) {
167     var i = 0,
168         l = 0;
169
170     // Clean canvas
171     ctx.fillStyle = '#000';
172     ctx.fillRect(0, 0, canvas.width, canvas.height);
173
174     // Draw player
175     //ctx.fillStyle = '#0f0';
176     ctx.strokeStyle = '#0f0';
177     for (i = 0, l = body.length; i < l; i += 1) {
178         //body[i].fill(ctx);
179         body[i].drawImage(ctx, iBody);
180     }
181
182     // Draw walls
183     //ctx.fillStyle = '#999';
184     //ctx.strokeStyle = '#999';
185     //for(i = 0 ,l = wall.length; i < l; i += 1) {
186     //    wall[i].fill(ctx);
187     //}
188
189     // Draw food
190     //ctx.fillStyle = '#f00';
191     //food.fill(ctx);
192     ctx.strokeStyle = '#f00';
193     food.drawImage(ctx, iFood);
194

```



```

195 // Debug last key pressed
196 ctx.fillStyle = '#fff';
197 //ctx.fillText('Last Press: ' + lastPress, 0, 20);
198
199 // Draw score
200 ctx.fillText('Score: ' + score, 0, 10);
201
202 // Draw pause
203 if (pause) {
204     ctx.textAlign = 'center';
205     if (gameover) {
206         ctx.fillText('GAME OVER', 150, 75);
207     } else {
208         ctx.fillText('PAUSE', 150, 75);
209     }
210     ctx.textAlign = 'left';
211 }
212 }
213
214 function act() {
215     var i = 0,
216         l = 0;
217
218     if (!pause) {
219         // GameOver Reset
220         if (gameover) {
221             reset();
222         }
223
224         // Move Body
225         for (i = body.length - 1; i > 0; i -= 1) {
226             body[i].x = body[i - 1].x;
227             body[i].y = body[i - 1].y;
228         }
229
230         // Change Direction
231         if (lastPress === KEY_UP && dir !== 2) {
232             dir = 0;
233         }
234         if (lastPress === KEY_RIGHT && dir !== 3) {
235             dir = 1;
236         }
237         if (lastPress === KEY_DOWN && dir !== 0) {
238             dir = 2;
239         }
240         if (lastPress === KEY_LEFT && dir !== 1) {
241             dir = 3;
242         }
243
244         // Move Head
245         if (dir === 0) {
246             body[0].y -= 10;
247         }
248         if (dir === 1) {
249             body[0].x += 10;
250         }
251         if (dir === 2) {
252             body[0].y += 10;
253         }
254         if (dir === 3) {
255             body[0].x -= 10;
256         }
257
258         // Out Screen
259         if (body[0].x > canvas.width - body[0].width) {
260             body[0].x = 0;
261         }
262         if (body[0].y > canvas.height - body[0].height) {
263             body[0].y = 0;
264         }
265         if (body[0].x < 0) {
266             body[0].x = canvas.width - body[0].width;

```

```

267     }
268     if (body[0].y < 0) {
269         body[0].y = canvas.height - body[0].height;
270     }
271
272     // Wall Intersects
273     //for(i = 0, l = wall.length; i < l; i += 1){
274     //    if (food.intersects(wall[i])) {
275     //        food.x = random(canvas.width / 10 - 1) * 10;
276     //        food.y = random(canvas.height / 10 - 1) * 10;
277     //    }
278     //
279     //    if(body[0].intersects(wall[i])){
280     //        gameover = true;
281     //        pause = true;
282     //    }
283     //}
284
285     // Body Intersects
286     for (i = 2, l = body.length; i < l; i += 1) {
287         if (body[0].intersects(body[i])) {
288             gameover = true;
289             pause = true;
290             aDie.play();
291         }
292     }
293
294     // Food Intersects
295     if (body[0].intersects(food)) {
296         body.push(new Rectangle(food.x, food.y, 10, 10));
297         score += 1;
298         food.x = random(canvas.width / 10 - 1) * 10;
299         food.y = random(canvas.height / 10 - 1) * 10;
300         aEat.play();
301     }
302 }
303
304 // Pause/Unpause
305 if (lastPress === KEY_ENTER) {
306     pause = !pause;
307     lastPress = null;
308 }
309 }
310
311 function repaint() {
312     window.requestAnimationFrame(repaint);
313     paint(ctx);
314 }
315
316 function run() {
317     setTimeout(run, 50);
318     act();
319 }
320
321 function init() {
322     // Get canvas and context
323     canvas = document.getElementById('canvas');
324     ctx = canvas.getContext('2d');
325
326     // Load assets
327     iBody.src = 'assets/body.png';
328     iFood.src = 'assets/fruit.png';
329     if (canPlayOgg()) {
330         aEat.src = 'assets/chomp.oga';
331         aDie.src = 'assets/dies.oga';
332     } else {
333         aEat.src = 'assets/chomp.m4a';
334         aDie.src = 'assets/dies.m4a';
335     }
336
337     // Create food
338     food = new Rectangle(80, 80, 10, 10);

```

```
339
340     // Create walls
341     //wall.push(new Rectangle(100, 50, 10, 10));
342     //wall.push(new Rectangle(100, 100, 10, 10));
343     //wall.push(new Rectangle(200, 50, 10, 10));
344     //wall.push(new Rectangle(200, 100, 10, 10));
345
346     // Start game
347     run();
348     repaint();
349 }
350
351 window.addEventListener('load', init, false);
352 }(window));
353
```

## 01.08. Snake - RequestAnimationFrame

En el pasado, para crear temporizadores en general, se utilizaba la función `setTimeout`. Esta función ha servido desde hace muchos años para toda clase de acciones por temporizador para las páginas web, sin embargo, no fue contemplada para animaciones, que requiere múltiples llamadas por segundo, consumiendo muchos recursos de nuestra computadora, aun si no estamos haciendo uso de la aplicación en cuestión.

Las compañías desarrolladoras de navegadores web han estado consciente de ello, y por tanto han ideado una mejor solución para esta tarea: la función `requestAnimationFrame`.

Esta función optimiza el uso de información, actualizándose de forma automática tan pronto el CPU le permite (Comúnmente, 60 cuadros por segundo en computadoras de escritorio), mejorando la capacidad del manejo de información en animaciones, consumiendo menos recursos, e incluso mandando a dormir el ciclo cuando la aplicación deja de tener enfoque, dando como resultado, un mejor manejo de las animaciones.

Para usar `requestAnimationFrame`, tan solo debes llamarle como la primer línea de una función, enviando como primer parámetro la misma función que la ha mandado a llamar, para que le llame de regreso después del tiempo de intervalo, tal como se muestra a continuación:

javascript

```
function run(){
  window.requestAnimationFrame(run);
  act();
  paint(ctx);
}
```

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

### Soporte para navegadores antiguos.

En el tiempo que esta entrada está siendo escrita, `requestAnimationFrame` es una función relativamente nueva, por lo que los navegadores que no estén actualizados podrían no soportarla, o usar una función experimental no-estándar de ella. Para saber más sobre la versión cuando esta función fue implementada, puedes visitar <http://caniuse.com/requestanimationframe>, dónde muestran el avance de su soporte.

Para poder usar `requestAnimationFrame` en estos navegadores antiguos, existen muchas soluciones posibles. La más simple y popular, es agregar esta función a tu código:

javascript

```
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. 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 viejas 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 nuestro juegos.

### Midiendo los cuadros por segundo.

Para comprobar el rendimiento real de `requestAnimationFrame` en nuestro dispositivo, calcularemos los cuadros por segundo (FPS) de nuestro lienzo. Para hacer esto, usaremos el código de [Parte 2. Animando el canvas](#). Empecemos creando cuatro variables, cuyos propósitos iré explicando conforme avancemos en la explicación:

javascript

```
var lastUpdate = 0,
    FPS = 0,
    frames = 0,
    acumDelta = 0;
```

Insertaremos el código a continuación dentro de la función `run`, justo después de llamar a `requestAnimationFrame`. Para empezar, calcularemos la delta del tiempo, esto es, el tiempo que ha pasado desde la última vez que el ciclo fue ejecutado en milisegundos, y lo dividiremos entre 1000 para convertirlo en segundos:

javascript

```
var now = Date.now(),
    deltaTime = (now - lastUpdate) / 1000;
if (deltaTime > 1) {
  deltaTime = 0;
}
lastUpdate = now;
```

En la primer línea, almacenamos el resultado de la función en una variable. Este valor es la cantidad de milisegundos desde el 1 enero de 1970, 00:00:00 UTC (Un estándar).

En la segunda línea, calculamos la delta del tiempo restando el ahora, al tiempo que teníamos almacenado. Para comprender esto mejor, hay que notar antes que en la cuarta línea, la variable `lastUpdate` obtiene el valor de `now`. Ahora sí, comprendemos que al siguiente ciclo, `now - lastUpdate` nos dará la delta del tiempo.

Sin embargo, en el primer ciclo, `lastUpdate` tiene un valor de cero, por lo que restar `now - lastUpdate` nos haría un valor enorme que causaría efectos no deseados. Por ello, en la tercer línea, si el tiempo delta es mayor a un segundo, su valor será descartado. Esto también sirve para prevenir valores no deseados en caso que el usuario salga de la pestaña actual por un momento, o la computadora se cuelgue por algunos segundos.

Después de comprender esta compleja pero simple sección, pasemos a lo que sigue. En cada ciclo sumaremos en uno los cuadros, y sumaremos la delta del tiempo a `acumDelta`:

```
frames += 1;
acumDelta += deltaTime;
```

javascript

Posteriormente, averiguaremos si ha pasado ya un segundo, preguntando si `acumDelta` es mayor a 1; de ser así, asignaremos la cantidad de cuadros pasados a nuestra variable `FPS`, y haremos un reset a las variables `frames` y `acumDelta`:

```
if (acumDelta > 1) {
    FPS = frames;
    frames = 0;
    acumDelta -= 1;
}
```

javascript

De esta forma, hemos obtenido los cuadros por segundo de nuestro juego. Ahora solo nos falta imprimirlos en pantalla:

```
ctx.fillText('FPS: ' + FPS, 10, 10);
```

javascript

La posible desventaja de usar `requestAnimationFrame`, es que tiene una tasa de actualización muy pequeña, 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 regularizar el tiempo, los cuales hay muchos, pero esos los veremos en la siguiente entrega.

## Código Final

```
1  /* RequestAnimationFrame */
2  'use strict';
3  var canvas = null,
4      ctx = null,
5      lastUpdate = 0,
6      FPS = 0,
7      frames = 0,
8      acumDelta = 0,
9      x = 50,
10     y = 50;
11
12 window.requestAnimationFrame = (function () {
13     return window.requestAnimationFrame ||
14         window.webkitRequestAnimationFrame ||
15         window.mozRequestAnimationFrame ||
16         function (callback) {
17             window.setTimeout(callback, 17);
18         };
19 })();
20
21 function paint(ctx) {
22     ctx.fillStyle = '#000';
23     ctx.fillRect(0, 0, canvas.width, canvas.height);
24
25     ctx.fillStyle = '#0f0';
26     ctx.fillRect(x, y, 10, 10);
27
28     ctx.fillStyle = '#fff';
```

javascript

```
29     ctx.fillText('FPS: ' + FPS, 10, 10);
30 }
31
32 function act() {
33     x += 2;
34     if (x > canvas.width) {
35         x = 0;
36     }
37 }
38
39 function run() {
40     window.requestAnimationFrame(run);
41
42     var now = Date.now(),
43         deltaTime = (now - lastUpdate) / 1000;
44     if (deltaTime > 1) {
45         deltaTime = 0;
46     }
47     lastUpdate = now;
48
49     frames += 1;
50     acumDelta += deltaTime;
51     if (acumDelta > 1) {
52         FPS = frames;
53         frames = 0;
54         acumDelta -= 1;
55     }
56
57     act();
58     paint(ctx);
59 }
60
61 function init() {
62     canvas = document.getElementById('canvas');
63     ctx = canvas.getContext('2d');
64     run();
65 }
66
67 window.addEventListener('load', init, false);
```

## 01.09. Snake - RequestAnimationFrame: Regulando el tiempo entre dispositivos

Debido a las diferentes capacidades en los dispositivos, requestAnimationFrame no es consistente entre ellos. Su tiempo de actualización varía en intervalos menores para dispositivos de menor poder, que hará los juegos más lento en unos dispositivos que en otros. Para evitar este problema, es necesario regular el tiempo entre estos dispositivos.

Existen varias formas de hacer esta acción. A continuación mostraré algunas de las formas mas conocidas y populares:

### 1. Envolver requestAnimationFrame en setTimeout.

Tan simple como se lee. Cuando busqué en Internet información de como regularizar el tiempo con requestAnimationFrame, esta fue la primer respuesta en aparecer. Se haría de la siguiente forma:

```
setTimeout( function () {  
    window.requestAnimationFrame(run)  
}, 50);
```

javascript

Esto es prácticamente hacer un setTimeout, con la ventaja de optimización de requestAnimationFrame. O al menos eso es lo que decía el sitio donde lo leí.

¿Efectivo? Lo dudo mucho. Al hacer la prueba a 50 milisegundos, el resultado me daba 15 cuadros por segundo en lugar de los 20 esperados, lo que me indica que este método es poco efectivo. Pero al menos, crea un tiempo regular entre dispositivos.

#### Ventajas:

- Fácil de implementar

#### Desventajas:

- No cumple el tiempo esperado en tiempos altos.
- Poco efectivo en tiempos bajos.

[Ve el ejemplo y código de este método.](#)

### 2. requestAnimationFrame para paint, setTimeout para act.

Una alternativa para regularizar el tiempo, 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 a veces querrás que algunos valores interactúen entre las funciones act y paint, lo que podría volverse una tarea un poco más compleja de lo que uno está acostumbrado.

La forma más sencilla de realizar un método asíncrono, es (como lo dice el título), 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 si mismas continuamente:

```
function repaint() {  
    window.requestAnimationFrame(repaint)  
    paint(ctx);  
}  
  
function run() {  
    setTimeout(run, 50);  
    act();  
}
```

javascript

Solo hay que llamar a ambas funciones (run y repaint) al final de la función init, y el juego correrá como de costumbre.

En el ejemplo, he agregado además de los cuadros por segundo, una segunda variable llamada ciclos por segundo (CPS), que demuestra que, efectivamente, ambas funciones cumplen con su objetivo.

#### Ventajas:

- Fácil de implementar.
- Efectivo en tiempos altos.

#### Desventajas:

- Es asíncrono
- Poco efectivo en tiempos bajos.
- Sigue consumiendo CPU si el juego pasa a segundo plano, aunque en mucha menor medida que al no usar requestAnimationFrame para paint.

[Ve el ejemplo y código de este método.](#)

### 3. Usar la delta de tiempo.

¡El método más popular y efectivo! Implementarlo será muy sencillo, ya que la parte difícil ya la has hecho antes, cuando implementamos los cuadros por segundo.

Prácticamente es tomar el mismo código para calcular el `deltaTime`, enviando este valor cuando llamas a la función `act`. Con este valor, solo debes multiplicarlo por el desplazamiento de los objetos, y de esta forma tendrás garantizado que siempre se desplazará la misma cantidad de píxeles por segundo, independiente de los cuadros por segundo en cada dispositivo.

¿Que valor se debe usar para esta multiplicación? La cantidad de píxeles que quieras se desplace tu objeto en un segundo. Para el ejemplo actual, estamos haciendo un desplazamiento de 2 píxeles por ciclo, en un tiempo óptimo de 60 cuadros por segundo... Eso quiere decir que nos estamos desplazando 120 píxeles por segundo:

javascript

```
function act(deltaTime) {  
  x += 120 * deltaTime;  
  if (x > canvas.width) {  
    x = 0;  
  }  
}
```

Para este ejemplo, este método ha sido muy sencillo de aplicar, pero cuando tenemos en cuenta que debemos hacer esta multiplicación por cada movimiento y animación en nuestro juego, para garantizar consistencia, descubriremos que este método puede ser un poco difícil de mantener. Aun así, sigue siendo el más efectivo de todas las opciones.

#### Ventajas:

- Siempre fiable y consistente
- Las transiciones siempre son tan fluidas como el CPU lo permita

#### Desventajas:

- Difícil de mantener
- Se congela el juego al pasar a segundo plano.

[Ve el ejemplo y código de este método.](#)

### Conclusión: ¿Qué método usar?

Todos los métodos tienen ventajas y desventajas entre ellos. En realidad, la elección del método a utilizar, depende de las necesidades de tu juego. Solamente si no recomendaré el primero... Pero el tercero es la mejor opción, y el segundo es una forma fácil de implementar, que suele trabajar bien en la mayoría de los casos.

Se te anima a experimentar con ellos, y ver cual es el que mejor se acomoda a tus necesidades.



## 01.10. Snake - Programación orientada a objetos

En programación se llama "objeto" a un conjunto de propiedades y métodos que definen su comportamiento. A un conjunto de características definidas en base al cual se crea un objeto, se llama "clase".

Los lenguajes de programación permiten crear clases personalizadas además de aquellas que vienen predefinidas. Para comprender mejor lo aquí explicado, crearemos de ejemplo una clase para objetos de tipo rectángulo.

JavaScript, a diferencia de otros lenguajes, no tiene clases como tal. Pero se pueden definir funciones que actúan como clases, tal como mostramos en el siguiente código:

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

javascript

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

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

javascript

De esta forma, hemos creado un objeto de tipo rectángulo, al cual especificamos sus propiedades x (50), y (50), ancho (100) y alto (60).

Sin embargo, 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 ello, es recomendado que se asigne un valor predefinido en caso que no se especifique el valor indicado:

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

javascript

En esta línea, asignamos a this.x uno de dos valores. Se comprueba mediante (x == null) ? si su valor es nulo o indefinido. Si es así, se asigna el valor antes de los dos puntos (0), y en caso contrario, se asigna el valor posterior a los dos puntos (x).

Esta asignación predefinida se repite para los demás valores. Personalmente, me gusta hacer algo distinto con la altura; en caso de que su valor sea nulo o indefinido, en lugar de asignarle 0, le asigno 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;
```

javascript

Nota que se asigna 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.

Antes de la programación orientada a objetos, cada propiedad de un objeto debía almacenarse en una variable independiente. Si bien esto no parece gran cosa para uno o dos objetos en el código, cuando se maneja decenas o cientos de estos, se vuelve una tarea complicada el no hacer estos programas sin la ayuda de objetos.

Otra propiedad importante de los objetos, es que tienen métodos propios. Por ejemplo, agregaremos este método a la clase Rectángulo, la cual nos permitirá saber cuando dicho rectángulo esté en intersección con otro:

```
this.intersects = function (rect) {  
  if (rect == null) {  
    window.console.warn('Missing parameters on function intersects');  
  } else {  
    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);  
  }  
};
```

javascript

Para llamar a este método, lo haríamos de la siguiente forma:

```
if (rect1.intersects(rect2)) {  
  
}
```

javascript

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 si el rectángulo es nulo, y si se da el caso, advertimos sobre el parámetro faltante, pues en caso de no enviar nada, si esta validación no se efectuara, provocaría un error. Por el contrario, si el rectángulo existe, se ejecuta una comparación de los cuatro puntos de ambos rectángulos, para comprobar si el contenido ambos rectángulos se intersecta 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. Mediante esta función, solo debemos indicarle el contexto 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 error en los valores dados:

javascript

```
this.fill = function (ctx) {  
  if (ctx == null) {  
    window.console.warn('Missing parameters on function fill');  
  } else {  
    ctx.fillRect(this.x, this.y, this.width, this.height);  
  }  
};
```

Usar rectángulos y otros objetos en los códigos de tus juegos, facilitarán muchas de las tareas que necesitas para llevarlos a cabo. Con esto, concluimos la explicación básica sobre programación orientada a objetos. Para conocer más a profundidad sobre objetos en JavaScript, puedes hacerlo en el siguiente enlace: [http://developer.mozilla.org/es/docs/Introducción\\_a\\_JavaScript\\_orientado\\_a\\_objetos](http://developer.mozilla.org/es/docs/Introducción_a_JavaScript_orientado_a_objetos).

### Código final:

javascript

```
1 function Rectangle(x, y, width, height) {  
2   this.x = (x == null) ? 0 : x;  
3   this.y = (y == null) ? 0 : y;  
4   this.width = (width == null) ? 0 : width;  
5   this.height = (height == null) ? this.width : height;  
6  
7   this.intersects = function (rect) {  
8     if (rect == null) {  
9       window.console.warn('Missing parameters on function intersects');  
10    } else {  
11      return (this.x < rect.x + rect.width &&  
12        this.x + this.width > rect.x &&  
13        this.y < rect.y + rect.height &&  
14        this.y + this.height > rect.y);  
15    }  
16  };  
17  
18  this.fill = function (ctx) {  
19    if (ctx == null) {  
20      window.console.warn('Missing parameters on function fill');  
21    } else {  
22      ctx.fillRect(this.x, this.y, this.width, this.height);  
23    }  
24  };  
25 }
```

## 01.11. Snake - Estirar el canvas y llenar la pantalla

Los que han creado proyectos antes en Flash, recordarán una de sus características singulares, que a veces podía ser funcional y otras veces molesto, dependiendo el efecto que se deseaba hacer. Hablo de la peculiar propiedad que, si ponías el contenedor de diferente tamaño al proyecto original, este se escalaba para ajustarse al nuevo tamaño.

Los HTML5 Canvas aparentan en un comienzo no tener esta propiedad, pues si cambiamos el ancho o alto de uno, solo nos da un escenario más grande o chico, pero este no se escala. Lo que ocurre, es que aprovecha una segunda propiedad para escalar el canvas, de forma aparte a poner su tamaño inicial. Lo hace mediante la etiqueta `Style`.

Para comprobar esto, tomaremos el juego de la serpiente, y buscaremos al comienzo donde creamos la etiqueta canvas que contiene nuestro juego. Lo modificaremos agregando en el atributo `style`, las propiedades `width` y `height`:

```
<canvas id="canvas" width="300" height="150" style="background:#999; width:600px; height:300px;">
  [Canvas not supported by your browser]
</canvas>
```

markup

Podemos comprobar que al comienzo, tenemos los atributos `width` y `height`:

```
width="300" height="150"
```

markup

Pero posteriormente, los tenemos como propiedad dentro del `style`, con valores al doble que en su atributo:

```
width:600px; height:300px;
```

CSS

Al abrir el archivo, notaremos que nuestro juego está ahora escalado al doble. De esta forma, es como podemos escalar un juego a diferentes tamaños.

### Llenar la pantalla.

Supondré que ahora que has visto como escalar un juego, has pensado en llenar la pantalla de tu navegador con él para aprovechar mejor los monitores de los usuarios. Lo primero que viene a nuestra mente, es darle ancho y alto del 100%, con el siguiente código:

```
<canvas id="canvas" width="300" height="150" style="background:#999; width:100%; height:100%;">
  [Canvas not supported by your browser]
</canvas>
```

markup

Pero al hacerlo, descubrimos dos problemas. El primero es que existe un pequeño margen predeterminado en nuestra página, por lo que realmente no estamos llenando nuestra pantalla con el lienzo. Para quitar este margen, solo debemos eliminar el `"margin"` y `"padding"` de nuestra etiqueta `"body"`:

```
<body style="margin:0; padding:0;">
```

markup

El segundo problema que descubrimos, es que la imagen se estira de forma no proporcionada. Y hay que tomar en cuenta que no todas las pantallas tienen la misma proporción, por lo que en pantallas más cuadradas se verá la imagen aun más achatada que en pantallas widescreen.

Para hacer que llene lo mayor posible la pantalla de forma proporcional tendremos que crear una función en Javascript, a la que llamaremos `"resize"`. Lo primero que hará esta función, es encontrar la proporción de nuestra pantalla, y determinar la escala a la cual debe ser estirado, dependiendo del ancho y alto de la pantalla:

```
var w = window.innerWidth / canvas.width;
var h = window.innerHeight / canvas.height;
var scale = Math.min(h, w);
```

javascript

Una vez obteniendo este valor, asignamos el ancho y alto al estilo de nuestro lienzo de acuerdo a la escala resultante:

```
canvas.style.width = (canvas.width * scale) + 'px';
canvas.style.height = (canvas.height * scale) + 'px';
```

javascript

Ahora tan solo debemos llamar a la función "resize" desde nuestra función "init", y tendremos nuestro juego llenando la máxima área disponible en el navegador. Dado que esta función solo se llama al comienzo del código, si el usuario cambia el tamaño de la ventana más tarde, el lienzo no se re-ajustará al nuevo tamaño. Podemos evitar ese problema, agregando un escucha al evento de cambio de tamaño del navegador, llamando de nuevo a la función en tal caso:

```
window.addEventListener('resize', resize, false);
```

javascript

Para complementar mejor este efecto, recomiendo que pongas el color de fondo de la página en negro modificando el estilo de la etiqueta "body", así como centrar su contenido, para el caso en que el lienzo extendido sea más alto que ancho:

```
<body style="margin:0; padding:0; background:#000; text-align:center">
```

markup

## 01.12. Snake - Doble búfer y escalado pixeleado

Es común que en los temas de desarrollo de videojuegos se vea sobre una técnica llamada doble búfer. La mayoría de los programas cuando dibujan en pantalla, lo hacen en tiempo real. Esto hace que uno pueda percibir como la pantalla se limpia y se vuelve a dibujar, causando un parpadeo molesto. Para evitar este efecto indeseado, la técnica consistía en dibujar toda la pantalla del juego en un lienzo oculto, y posteriormente dibujar este lienzo final sobre el lienzo que se muestra al usuario. Esta es la técnica conocida bajo el nombre de doble búfer.

Sin embargo, la tecnología de HTML5 Canvas parece ya manejar esta técnica por detrás, por lo que ya no resulta necesario implementarla como en otros lenguajes.

Aun cuando este es el propósito principal del doble búfer, no es la única razón para la que es ocupada. Muchos efectos visuales avanzados como sustracción de colores, iluminación, distorsión y brillo se ejecutan sobre un búfer oculto antes de mostrarse al usuario, sobre todo en juegos 3D.

El día de hoy, aprenderemos como usar el doble búfer para hacer un efecto muy sencillo, que es el escalado pixeleado, el cual nos permitirá crear juegos con aspecto retro para pantallas grandes con HTML5 Canvas.

Para empezar, declararemos las variables que contendrán nuestro buffer y su contexto:

```
var buffer = null,
    bufferCtx = null;
```

javascript

Dentro de la función "init", creamos de forma dinámica un elemento canvas que asignamos a la variable "buffer". Después asignamos a "bufferCtx" el contexto 2D de nuestro búfer, y finalmente asignamos el ancho y alto predefinido a nuestro búfer:

```
// Load buffer
buffer = document.createElement('canvas');
bufferCtx = buffer.getContext('2d');
buffer.width = 300;
buffer.height = 150;
```

javascript

Finalmente, para dibujar nuestro búfer en el lienzo principal, hemos de modificar la función "repaint" de tal forma que la función "paint" envíe el contexto del búfer en lugar del contexto principal. Una vez que nuestro juego haya sido dibujado en el búfer, proseguimos a dibujar el lienzo principal limpiando la pantalla, y dibujando el resultado de nuestro búfer como una imagen, asignándole el ancho y alto del lienzo principal:

```
function repaint() {
    window.requestAnimationFrame(repaint);
    paint(bufferCtx);

    ctx.fillStyle = '#000';
    ctx.fillRect(0, 0, canvas.width, canvas.height);
    ctx.drawImage(buffer, 0, 0, canvas.width, canvas.height);
}
```

javascript

Dado que ahora el tamaño de nuestro lienzo virtual en el juego es el del búfer y no el del lienzo principal, no olvides convertir todas las referencias a "canvas" dentro de las funciones "run" y "paint" por referencias a "buffer"; especialmente aquellas que obtienen los valores de su ancho y alto para definir los límites del escenario.

Ahora que nuestro doble buffer esta listo, es tiempo de realizar la magia que hemos esperado desde el comienzo de esta entrada. De forma predefinida, el suavizado de imágenes escaladas está activado, por lo que para poder dibujar imágenes con efecto pixeleado, necesitamos desactivar dicho suavizado. Para hacer dicho efecto, agregamos esta línea a nuestro contexto principal dentro de la función "repaint", justo antes de dibujar nuestro búfer en el lienzo principal:

```
ctx.imageSmoothingEnabled = false;
```

javascript

Así es, toda la complejidad de este efecto se reduce a una línea. Quizá ahora te preguntes ¿Por qué no simplemente asignamos esa línea desde el comienzo y nos evitamos todas las complicaciones del doble búfer? Pues bien, resulta que esta línea solo afecta a los dibujos escalados por el contexto que los dibuja, por lo que escalar el lienzo con CSS como lo hemos hecho antes no se vería afectado por esta línea.

La alternativa que se tendría a escalar el búfer de esta forma, sería escalar cada uno de los elementos dentro de nuestro lienzo, lo cual no solo sería complicado al tener que recalcular la posición y escala de cada elemento, si no que además impactaría de forma negativa al procesador, lo que podría hacer a nuestro juego muy lento en caso de ser ya un proyecto grande y complejo.

Podemos ver el resultado de este código a continuación:

Dado que el parámetro `imageSmoothingEnabled` aun no es un estándar, es necesario agregar además las versiones con prefijo adecuados para cada motor:

```
ctx.webkitImageSmoothingEnabled = false;
ctx.mozImageSmoothingEnabled = false;
ctx.msImageSmoothingEnabled = false;
ctx.oImageSmoothingEnabled = false;
```

javascript

### Doble búfer y llenado de pantalla

Otra ventaja del doble búfer es la facilidad con la que se puede aprovechar al máximo el área donde se está dibujando. Centrar el contenido de nuestro búfer es bastante sencillo. Para empezar, necesitamos almacenar la escala del búfer, así como la compensación de distancia con respecto al centro:

```
var bufferScale = 1,
    bufferOffsetX = 0,
    bufferOffsetY = 0;
```

javascript

Creemos una función `resize` como vimos en el tema [Estirar el canvas y llenar la pantalla](#). Al comienzo, asignaremos el tamaño de nuestro lienzo principal al tamaño completo de la pantalla. Posteriormente, obtendremos la escala del búfer de la misma forma que aprendimos en la entrada anterior, y finalmente, calculamos la compensación mediante la diferencia del lienzo principal y el buffer escalado, dividido entre dos:

```
function resize() {
    canvas.width = window.innerWidth;
    canvas.height = window.innerHeight;

    var w = window.innerWidth / buffer.width;
    var h = window.innerHeight / buffer.height;
    bufferScale = Math.min(h, w);

    bufferOffsetX = (canvas.width - (buffer.width * bufferScale)) / 2;
    bufferOffsetY = (canvas.height - (buffer.height * bufferScale)) / 2;
}
```

javascript

No olvides llamar la función "resize" tanto al inicio, como en el escucha de cambio de tamaño. Finalmente, para ver reflejados estos cambios, actualiza la función `repaint` para dibujar el buffer con los parámetros correspondientes:

```
ctx.drawImage(buffer, bufferOffsetX, bufferOffsetY, buffer.width * bufferScale, buffer.height * buf
```

javascript

Puedes ver el resultado final en [canvas.ninja/?js=snake-pixel](https://canvas.ninja/?js=snake-pixel), donde podrás comprobar que, independiente del tamaño del navegador, el juego se encontrará siempre centrado, tanto de forma horizontal como vertical.

De esta forma, es como hemos aprendido como mantener un lienzo con estilo pixelado después de escalarlo gracias a un doble búfer.

### Código final:

```
1  /*jslint bitwise:true, es5: true */
2  (function (window, undefined) {
3      'use strict';
4      var KEY_ENTER = 13,
5          KEY_LEFT = 37,
6          KEY_UP = 38,
7          KEY_RIGHT = 39,
8          KEY_DOWN = 40,
9
10     canvas = null,
```

javascript

```

11     ctx = null,
12     buffer = null,
13     bufferCtx = null,
14     bufferScale = 1,
15     bufferOffsetX = 0,
16     bufferOffsetY = 0,
17     lastPress = null,
18     pause = true,
19     gameover = true,
20     fullscreen = false,
21     body = [],
22     food = null,
23     //var wall = [],
24     dir = 0,
25     score = 0,
26     iBody = new Image(),
27     iFood = new Image(),
28     aEat = new Audio(),
29     aDie = new Audio();
30
31 window.requestAnimationFrame = (function () {
32     return window.requestAnimationFrame ||
33         window.mozRequestAnimationFrame ||
34         window.webkitRequestAnimationFrame ||
35         function (callback) {
36             window.setTimeout(callback, 17);
37         };
38 }());
39
40 document.addEventListener('keydown', function (evt) {
41     if (evt.which >= 37 && evt.which <= 40) {
42         evt.preventDefault();
43     }
44
45     lastPress = evt.which;
46 }, false);
47
48 function Rectangle(x, y, width, height) {
49     this.x = (x === undefined) ? 0 : x;
50     this.y = (y === undefined) ? 0 : y;
51     this.width = (width === undefined) ? 0 : width;
52     this.height = (height === undefined) ? this.width : height;
53 }
54
55 Rectangle.prototype = {
56     constructor: Rectangle,
57
58     intersects: function (rect) {
59         if (rect === undefined) {
60             window.console.warn('Missing parameters on function intersects');
61         } else {
62             return (this.x < rect.x + rect.width &&
63                 this.x + this.width > rect.x &&
64                 this.y < rect.y + rect.height &&
65                 this.y + this.height > rect.y);
66         }
67     },
68
69     fill: function (ctx) {
70         if (ctx === undefined) {
71             window.console.warn('Missing parameters on function fill');
72         } else {
73             ctx.fillRect(this.x, this.y, this.width, this.height);
74         }
75     },
76
77     drawImage: function (ctx, img) {
78         if (img === undefined) {
79             window.console.warn('Missing parameters on function drawImage');
80         } else {
81             if (img.width) {
82                 ctx.drawImage(img, this.x, this.y);

```

```

83         } else {
84             ctx.strokeRect(this.x, this.y, this.width, this.height);
85         }
86     }
87 }
88 };
89
90 function random(max) {
91     return ~~(Math.random() * max);
92 }
93
94 function resize() {
95     canvas.width = window.innerWidth;
96     canvas.height = window.innerHeight;
97
98     var w = window.innerWidth / buffer.width;
99     var h = window.innerHeight / buffer.height;
100     bufferScale = Math.min(h, w);
101
102     bufferOffsetX = (canvas.width - (buffer.width * bufferScale)) / 2;
103     bufferOffsetY = (canvas.height - (buffer.height * bufferScale)) / 2;
104 }
105
106 function reset() {
107     score = 0;
108     dir = 1;
109     body.length = 0;
110     body.push(new Rectangle(40, 40, 10, 10));
111     body.push(new Rectangle(0, 0, 10, 10));
112     body.push(new Rectangle(0, 0, 10, 10));
113     food.x = random(buffer.width / 10 - 1) * 10;
114     food.y = random(buffer.height / 10 - 1) * 10;
115     gameover = false;
116 }
117
118 function paint(ctx) {
119     var i = 0,
120         l = 0;
121
122     // Clean canvas
123     ctx.fillStyle = '#030';
124     ctx.fillRect(0, 0, buffer.width, buffer.height);
125
126     // Draw player
127     ctx.strokeStyle = '#0f0';
128     for (i = 0, l = body.length; i < l; i += 1) {
129         body[i].drawImage(ctx, iBody);
130     }
131
132     // Draw walls
133     //ctx.fillStyle = '#999';
134     //for (i = 0, l = wall.length; i < l; i += 1) {
135     //    wall[i].fill(ctx);
136     //}
137
138     // Draw food
139     ctx.strokeStyle = '#f00';
140     food.drawImage(ctx, iFood);
141
142     // Draw score
143     ctx.fillStyle = '#fff';
144     ctx.fillText('Score: ' + score, 0, 10);
145
146     // Debug last key pressed
147     //ctx.fillText('Last Press: '+lastPress,0,20);
148
149     // Draw pause
150     if (pause) {
151         ctx.textAlign = 'center';
152         if (gameover) {
153             ctx.fillText('GAME OVER', 150, 75);
154         } else {

```



```

155         ctx.fillText('PAUSE', 150, 75);
156     }
157     ctx.textAlign = 'left';
158 }
159 }
160
161 function act() {
162     var i = 0,
163         l = 0;
164
165     if (!pause) {
166         // GameOver Reset
167         if (gameover) {
168             reset();
169         }
170
171         // Move Body
172         for (i = body.length - 1; i > 0; i -= 1) {
173             body[i].x = body[i - 1].x;
174             body[i].y = body[i - 1].y;
175         }
176
177         // Change Direction
178         if (lastPress === KEY_UP && dir !== 2) {
179             dir = 0;
180         }
181         if (lastPress === KEY_RIGHT && dir !== 3) {
182             dir = 1;
183         }
184         if (lastPress === KEY_DOWN && dir !== 0) {
185             dir = 2;
186         }
187         if (lastPress === KEY_LEFT && dir !== 1) {
188             dir = 3;
189         }
190
191         // Move Head
192         if (dir === 0) {
193             body[0].y -= 10;
194         }
195         if (dir === 1) {
196             body[0].x += 10;
197         }
198         if (dir === 2) {
199             body[0].y += 10;
200         }
201         if (dir === 3) {
202             body[0].x -= 10;
203         }
204
205         // Out Screen
206         if (body[0].x > canvas.width - body[0].width) {
207             body[0].x = 0;
208         }
209         if (body[0].y > canvas.height - body[0].height) {
210             body[0].y = 0;
211         }
212         if (body[0].x < 0) {
213             body[0].x = canvas.width - body[0].width;
214         }
215         if (body[0].y < 0) {
216             body[0].y = canvas.height - body[0].height;
217         }
218
219         // Food Intersects
220         if (body[0].intersects(food)) {
221             body.push(new Rectangle(0, 0, 10, 10));
222             score += 1;
223             food.x = random(buffer.width / 10 - 1) * 10;
224             food.y = random(buffer.height / 10 - 1) * 10;
225             aEat.play();
226         }

```

```

227
228 // Wall Intersects
229 //for (i = 0, l = wall.length; i < l; i += 1) {
230 //    if (food.intersects(wall[i])) {
231 //        food.x = random(canvas.width / 10 - 1) * 10;
232 //        food.y = random(canvas.height / 10 - 1) * 10;
233 //    }
234 //
235 //    if (body[0].intersects(wall[i])) {
236 //        gameover = true;
237 //        pause = true;
238 //    }
239 //}
240
241 // Body Intersects
242 for (i = 2, l = body.length; i < l; i += 1) {
243     if (body[0].intersects(body[i])) {
244         gameover = true;
245         pause = true;
246         aDie.play();
247     }
248 }
249 }
250 // Pause/Unpause
251 if (lastPress === KEY_ENTER) {
252     pause = !pause;
253     lastPress = null;
254 }
255 }
256
257 function repaint() {
258     window.requestAnimationFrame(repaint);
259     paint(bufferCtx);
260
261     ctx.fillStyle = '#000';
262     ctx.fillRect(0, 0, canvas.width, canvas.height);
263     ctx.imageSmoothingEnabled = false;
264     ctx.drawImage(buffer, bufferOffsetX, bufferOffsetY, buffer.width * bufferScale, buffer.height);
265 }
266
267 function run() {
268     setTimeout(run, 50);
269     act();
270 }
271
272 function init() {
273     // Get canvas and context
274     canvas = document.getElementById('canvas');
275     ctx = canvas.getContext('2d');
276     canvas.width = 600;
277     canvas.height = 300;
278
279     // Load buffer
280     buffer = document.createElement('canvas');
281     bufferCtx = buffer.getContext('2d');
282     buffer.width = 300;
283     buffer.height = 150;
284
285     // Load assets
286     iBody.src = 'assets/body.png';
287     iFood.src = 'assets/fruit.png';
288     aEat.src = 'assets/chomp.m4a';
289     aDie.src = 'assets/dies.m4a';
290
291     // Create food
292     food = new Rectangle(80, 80, 10, 10);
293
294     // Create walls
295     //wall.push(new Rectangle(50, 50, 10, 10));
296     //wall.push(new Rectangle(50, 100, 10, 10));
297     //wall.push(new Rectangle(100, 50, 10, 10));
298     //wall.push(new Rectangle(100, 100, 10, 10));

```

```
299
300     // Start game
301     resize();
302     run();
303     repaint();
304 }
305
306 window.addEventListener('load', init, false);
307 window.addEventListener('resize', resize, false);
308 }(window));
```

## 01.13. Snake - Manejo de escenas

Ahora que ya sabemos hacer un juego, es tiempo de pensar en agregarle detalles para hacerlo más atractivo y funcional, y para ello necesitaremos múltiples escenas que aparecerán antes y después de la función principal de nuestro juego.

Existen varias formas de manejar escenas. Quizá la forma mas sencilla y directa de hacerlo, sea creando estados condicionales, como se muestra en el siguiente ejemplo:

javascript

```
var SCENE_MAIN = 0,
    SCENE_GAME = 1,

    currentScene = 0;

function act() {
    if (currentScene === SCENE_MAIN) {
        //...
    } else if (currentScene === SCENE_GAME) {
        //...
    }
}

function paint(ctx) {
    if (currentScene === SCENE_MAIN) {
        //...
    } else if (currentScene === SCENE_GAME) {
        //...
    }
}
```

Dentro de cada área condicional se pondría el código a ejecutar para el caso de cada escena deseada, cambiando el valor de la variable "currentScene" para simular el cambio de escena. Esta es una técnica bastante práctica, sin embargo, cuando los juegos son más complejos y con muchas escenas posibles, mantener un código así puede ser bastante difícil.

Es por ello que les enseñaré en esta ocasión una técnica diferente, que si bien es un poco más compleja de comprender al comienzo, al final resultará mucho mas práctica, automatizada y fácil de mantener.

Para empezar, necesitaremos un arreglo que almacene nuestras escenas, y la variable currentScene del ejemplo pasado para manejar las distintas escenas:

javascript

```
var currentScene = 0,
    scenes = [];
```

Las escenas las manejaremos a través de una función "Scene", la cual al momento de ser creada, le asignaremos automáticamente una ID única basada en el tamaño del arreglo de escenas, y lo pondremos dentro del mismo:

javascript

```
function Scene() {
    this.id = scenes.length;
    scenes.push(this);
}
```

Al prototipo de esta función le asignaremos las funciones que todas escenas manejaran: una función "act" y una función "paint" como la que ya hemos aprendido a manejar hasta ahora, además de una función "load" que se ejecutará automáticamente al cargar dicha escena:

javascript

```
Scene.prototype = {
    constructor: Scene,
    load: function () {},
    paint: function (ctx) {},
    act: function () {}
};
```

Si vas a manejar delta de tiempo, no olvides mandarla también a la función "act". Finalmente, agregaremos una función "loadScene" para que cargue nuestra nueva escena de forma automática, de esta forma:

javascript

```
function loadScene(scene) {
    currentScene = scene.id;
    scenes[currentScene].load();
}
```

Por último, modificaremos nuestras funciones "run" y "repaint" para que ejecute automáticamente la escena en la que nos encontramos:

```
function repaint() {
    window.requestAnimationFrame(repaint);
    if (scenes.length) {
        scenes[currentScene].paint(ctx);
    }
}

function run() {
    setTimeout(run, 50);
    if (scenes.length) {
        scenes[currentScene].act();
    }
}
```

Podrás notar que antes de ejecutar su respectiva función, comprobamos antes que haya al menos una escena dentro del arreglo de escenas. De esta forma, si olvidamos crear al menos una escena, no retornará un error. Ahora que ya tenemos todo listo, procedamos a declarar nuestras escenas. Para este ejemplo práctico, crearemos dos: la escena del menú principal, y la escena de nuestro juego:

```
var mainScene = null,
    gameScene = null;
```

Para que nuestro juego funcione en este nuevo formato, tenemos que cambiar nuestras funciones "act" y "paint" por "gameScene.act" y "gameScene.paint". Dado que estas funciones son variables ya existentes, tenemos que "re-asignarles" las funciones, de la siguiente forma:

```
// Game Scene
gameScene = new Scene();

//function reset(){
gameScene.load = function () {
    //...
}

//function paint(ctx){
gameScene.paint = function (ctx) {
    //...
}

//function act(){
gameScene.act = function () {
    //...
}
```

Notarás que también he cambiado la función "reset" por "gameScene.load" para que cargue automáticamente cuando se cambie a esta escena.

Ahora, agregaremos nuestro menú principal:

```
// Main Scene
mainScene = new Scene();

mainScene.paint = function (ctx) {
    // Clean canvas
    ctx.fillStyle = '#030';
    ctx.fillRect(0, 0, canvas.width, canvas.height);

    // Draw title
    ctx.fillStyle = '#fff';
    ctx.textAlign = 'center';
    ctx.fillText('SNAKE', 150, 60);
    ctx.fillText('Press Enter', 150, 90);
};

mainScene.act = function () {
    // Load next scene
    if (lastPress === KEY_ENTER) {
        loadScene(gameScene);
        lastPress = null;
    }
};
```

En esta escena, solo he agregado un escucha a la tecla "Enter" para que empiece el juego, y en la función "paint" se imprime tan solo el título del juego. Nota que no he asignado la función "load" a "mainScene", esto por que no es necesario cargar nada en esta escena. Sin embargo, la

escena que cargue predeterminadamente, no cargará su función "load" por defecto, y si se manda a llamar a ella antes de ser creada, no tendrá efecto alguno, por lo que si deseamos que la escena predeterminada cargue su función "load" al comienzo del juego, se tendrá que forzar su carga tras su creación de esta forma:

```
mainScene.load = function () {  
    //...  
}  
mainScene.load();
```

javascript

Para finalizar, ahora que ya tenemos todo nuestro juego con escenas preparado, cambiaremos el comportamiento en nuestro juego al hacer Game Over, para que nos regrese a la pantalla principal:

```
// GameOver Reset  
if (gameover) {  
    loadScene(mainScene);  
}
```

javascript

Con esto nuestro juego está listo para soportar múltiples escenas. Ya solo es necesario que tú crees y personalices las necesarias para tu propio juego. ¡Felices códigos!

### Código final:

```
1  /*jslint bitwise:true, es5: true */  
2  (function (window, undefined) {  
3      'use strict';  
4      var KEY_ENTER = 13,  
5          KEY_LEFT = 37,  
6          KEY_UP = 38,  
7          KEY_RIGHT = 39,  
8          KEY_DOWN = 40,  
9  
10     canvas = null,  
11     ctx = null,  
12     lastPress = null,  
13     pause = false,  
14     gameover = false,  
15     currentScene = 0,  
16     scenes = [],  
17     mainScene = null,  
18     gameScene = null,  
19     body = [],  
20     food = null,  
21     //var wall = [],  
22     dir = 0,  
23     score = 0,  
24     iBody = new Image(),  
25     iFood = new Image(),  
26     aEat = new Audio(),  
27     aDie = new Audio();  
28  
29     window.requestAnimationFrame = (function () {  
30         return window.requestAnimationFrame ||  
31             window.mozRequestAnimationFrame ||  
32             window.webkitRequestAnimationFrame ||  
33             function (callback) {  
34                 window.setTimeout(callback, 17);  
35             };  
36     })();  
37  
38     document.addEventListener('keydown', function (evt) {  
39         if (evt.which >= 37 && evt.which <= 40) {  
40             evt.preventDefault();  
41         }  
42     });
```

javascript

```

42
43     lastPress = evt.which;
44 }, false);
45
46 function Rectangle(x, y, width, height) {
47     this.x = (x === undefined) ? 0 : x;
48     this.y = (y === undefined) ? 0 : y;
49     this.width = (width === undefined) ? 0 : width;
50     this.height = (height === undefined) ? this.width : height;
51 }
52
53 Rectangle.prototype = {
54     constructor: Rectangle,
55
56     intersects: function (rect) {
57         if (rect === undefined) {
58             window.console.warn('Missing parameters on function intersects');
59         } else {
60             return (this.x < rect.x + rect.width &&
61                 this.x + this.width > rect.x &&
62                 this.y < rect.y + rect.height &&
63                 this.y + this.height > rect.y);
64         }
65     },
66
67     fill: function (ctx) {
68         if (ctx === undefined) {
69             window.console.warn('Missing parameters on function fill');
70         } else {
71             ctx.fillRect(this.x, this.y, this.width, this.height);
72         }
73     },
74
75     drawImage: function (ctx, img) {
76         if (img === undefined) {
77             window.console.warn('Missing parameters on function drawImage');
78         } else {
79             if (img.width) {
80                 ctx.drawImage(img, this.x, this.y);
81             } else {
82                 ctx.strokeRect(this.x, this.y, this.width, this.height);
83             }
84         }
85     }
86 };
87
88 function Scene() {
89     this.id = scenes.length;
90     scenes.push(this);
91 }
92
93 Scene.prototype = {
94     constructor: Scene,
95     load: function () {},
96     paint: function (ctx) {},
97     act: function () {}
98 };
99
100 function loadScene(scene) {
101     currentScene = scene.id;
102     scenes[currentScene].load();
103 }
104
105 function random(max) {
106     return ~~(Math.random() * max);
107 }
108
109 function repaint() {
110     window.requestAnimationFrame(repaint);
111     if (scenes.length) {
112         scenes[currentScene].paint(ctx);
113     }

```

```

114 }
115
116 function run() {
117     setTimeout(run, 50);
118     if (scenes.length) {
119         scenes[currentScene].act();
120     }
121 }
122
123 function init() {
124     // Get canvas and context
125     canvas = document.getElementById('canvas');
126     ctx = canvas.getContext('2d');
127
128     // Load assets
129     iBody.src = 'assets/body.png';
130     iFood.src = 'assets/fruit.png';
131     aEat.src = 'assets/chomp.m4a';
132     aDie.src = 'assets/dies.m4a';
133
134     // Create food
135     food = new Rectangle(80, 80, 10, 10);
136
137     // Create walls
138     //wall.push(new Rectangle(50, 50, 10, 10));
139     //wall.push(new Rectangle(50, 100, 10, 10));
140     //wall.push(new Rectangle(100, 50, 10, 10));
141     //wall.push(new Rectangle(100, 100, 10, 10));
142
143     // Start game
144     run();
145     repaint();
146 }
147
148 // Main Scene
149 mainScene = new Scene();
150
151 mainScene.paint = function (ctx) {
152     // Clean canvas
153     ctx.fillStyle = '#030';
154     ctx.fillRect(0, 0, canvas.width, canvas.height);
155
156     // Draw title
157     ctx.fillStyle = '#fff';
158     ctx.textAlign = 'center';
159     ctx.fillText('SNAKE', 150, 60);
160     ctx.fillText('Press Enter', 150, 90);
161 };
162
163 mainScene.act = function () {
164     // Load next scene
165     if (lastPress === KEY_ENTER) {
166         loadScene(gameScene);
167         lastPress = null;
168     }
169 };
170
171 // Game Scene
172 gameScene = new Scene();
173
174 gameScene.load = function () {
175     score = 0;
176     dir = 1;
177     body.length = 0;
178     body.push(new Rectangle(40, 40, 10, 10));
179     body.push(new Rectangle(0, 0, 10, 10));
180     body.push(new Rectangle(0, 0, 10, 10));
181     food.x = random(canvas.width / 10 - 1) * 10;
182     food.y = random(canvas.height / 10 - 1) * 10;
183     gameover = false;
184 };
185

```



```

186 gameScene.paint = function (ctx) {
187     var i = 0,
188         l = 0;
189
190     // Clean canvas
191     ctx.fillStyle = '#030';
192     ctx.fillRect(0, 0, canvas.width, canvas.height);
193
194     // Draw player
195     ctx.strokeStyle = '#0f0';
196     for (i = 0, l = body.length; i < l; i += 1) {
197         body[i].drawImage(ctx, iBody);
198     }
199
200     // Draw walls
201     //ctx.fillStyle = '#999';
202     //for (i = 0, l = wall.length; i < l; i += 1) {
203     //    wall[i].fill(ctx);
204     //}
205
206     // Draw food
207     ctx.strokeStyle = '#f00';
208     food.drawImage(ctx, iFood);
209
210     // Draw score
211     ctx.fillStyle = '#fff';
212     ctx.textAlign = 'left';
213     ctx.fillText('Score: ' + score, 0, 10);
214
215     // Debug last key pressed
216     //ctx.fillText('Last Press: '+lastPress,0,20);
217
218     // Draw pause
219     if (pause) {
220         ctx.textAlign = 'center';
221         if (gameover) {
222             ctx.fillText('GAME OVER', 150, 75);
223         } else {
224             ctx.fillText('PAUSE', 150, 75);
225         }
226     }
227 };
228
229 gameScene.act = function () {
230     var i = 0,
231         l = 0;
232
233     if (!pause) {
234         // GameOver Reset
235         if (gameover) {
236             loadScene(mainScene);
237         }
238
239         // Move Body
240         for (i = body.length - 1; i > 0; i -= 1) {
241             body[i].x = body[i - 1].x;
242             body[i].y = body[i - 1].y;
243         }
244
245         // Change Direction
246         if (lastPress === KEY_UP && dir !== 2) {
247             dir = 0;
248         }
249         if (lastPress === KEY_RIGHT && dir !== 3) {
250             dir = 1;
251         }
252         if (lastPress === KEY_DOWN && dir !== 0) {
253             dir = 2;
254         }
255         if (lastPress === KEY_LEFT && dir !== 1) {
256             dir = 3;
257         }

```

```

258
259 // Move Head
260 if (dir === 0) {
261     body[0].y -= 10;
262 }
263 if (dir === 1) {
264     body[0].x += 10;
265 }
266 if (dir === 2) {
267     body[0].y += 10;
268 }
269 if (dir === 3) {
270     body[0].x -= 10;
271 }
272
273 // Out Screen
274 if (body[0].x > canvas.width - body[0].width) {
275     body[0].x = 0;
276 }
277 if (body[0].y > canvas.height - body[0].height) {
278     body[0].y = 0;
279 }
280 if (body[0].x < 0) {
281     body[0].x = canvas.width - body[0].width;
282 }
283 if (body[0].y < 0) {
284     body[0].y = canvas.height - body[0].height;
285 }
286
287 // Food Intersects
288 if (body[0].intersects(food)) {
289     body.push(new Rectangle(0, 0, 10, 10));
290     score += 1;
291     food.x = random(canvas.width / 10 - 1) * 10;
292     food.y = random(canvas.height / 10 - 1) * 10;
293     aEat.play();
294 }
295
296 // Wall Intersects
297 //for (i = 0, l = wall.length; i < l; i += 1) {
298 //    if (food.intersects(wall[i])) {
299 //        food.x = random(canvas.width / 10 - 1) * 10;
300 //        food.y = random(canvas.height / 10 - 1) * 10;
301 //    }
302 //}
303 //    if (body[0].intersects(wall[i])) {
304 //        gameover = true;
305 //        pause = true;
306 //    }
307 //}
308
309 // Body Intersects
310 for (i = 2, l = body.length; i < l; i += 1) {
311     if (body[0].intersects(body[i])) {
312         gameover = true;
313         pause = true;
314         aDie.play();
315     }
316 }
317
318 // Pause/Unpause
319 if (lastPress === KEY_ENTER) {
320     pause = !pause;
321     lastPress = null;
322 }
323 };
324
325 window.addEventListener('load', init, false);
326 }(window));

```

## 01.14. Snake - Almacenamiento local y altos puntajes

Existen veces que necesitamos almacenar información del jugador dentro de su computadora. Por ejemplo, en juegos muy grandes de varios niveles, es posible que queramos conocer el nivel en el que se quedó nuestro jugador.

Para almacenar un dato en el navegador del usuario, utilizamos el Almacenamiento Local. Usarlo es bastante sencillo, únicamente debemos escribir `localStorage`, un punto, y el nombre de la variable con la que queremos guardar nuestro dato. Por ejemplo, para guardar el nivel en el que está nuestro jugador, escribimos esta línea:

```
localStorage.level = level; javascript
```

Este código sería buena idea ejecutarlo al pasar un nuevo nivel. Para recuperar ese dato, únicamente debemos comparar, en la función `init`, si el almacenamiento local tiene un valor en dicha variable, y de ser cierto, la asignamos a la variable de nuestro juego:

```
if (localStorage.level) { javascript  
    level = localStorage.level;  
}
```

Aun si cierras el navegador, apagas la computadora y regresas al día siguiente, los datos seguirán ahí disponibles para que continúes el juego donde lo dejaste.

Para ejemplificar de forma más clara las posibilidades que se tienen con el Almacenamiento Local, les mostraré como crear una lista de altos puntajes, almacenarla en el almacenamiento local, y recuperarla. Comencemos creando el arreglo que contendrá nuestros altos puntajes, y una variable que contendrá la posición de nuestro nuevo mejor puntaje:

```
var highscores = [], javascript  
    posHighscore = 10;
```

Para agregar un nuevo dato, se creará una función la cual será llamada al momento de que acabe nuestro juego (Por ejemplo, cuando el jugador pierda), a la cual enviaremos el puntaje del jugador, e intentará agregarlo a los mejores puntajes. Esta función será de la siguiente forma:

```
function addHighscore(score) { javascript  
    posHighscore = 0;  
    while (highscores[posHighscore] > score && posHighscore < highscores.length) {  
        posHighscore += 1;  
    }  
    highscores.splice(posHighscore, 0, score);  
    if (highscores.length > 10) {  
        highscores.length = 10;  
    }  
    localStorage.highscores = highscores.join(',');  
}
```

Para comprender mejor esta función, la explicaré por partes. Comenzamos buscando la posición de nuestro nuevo alto puntaje, asignándole el valor de cero, y mientras puntaje en la posición actual sea mayor al puntaje actual, o la posición alcance el final de la longitud de los altos puntajes, su valor se irá sumando en uno. Así se encontrará donde va nuestro nuevo puntaje en la lista. Una vez localizada la posición, se inserta al arreglo mediante la función "splice", mandando la posición, el valor de elementos a eliminar (0), y el nuevo valor a insertar (el puntaje).

Cuando hemos insertado el nuevo valor, verificaremos si la longitud del arreglo es mayor a 10, y de ser así, recortamos su longitud a solo 10 elementos. Así, mantendremos solo los 10 mejores puntajes almacenados.

Por último, guardamos nuestros puntajes en el almacenamiento local. Como el almacenamiento local no admite arreglos, primero debemos convertir el arreglo a texto. Esto se hace de forma muy sencilla con la función "join", que une todos los elementos del arreglo con una coma, quedando nuestro arreglo de una forma similar a esta:

```
"27,23,21,19,15,12,11,9,8,7" javascript
```

Para recuperar ahora los datos de los altos puntajes, simplemente separamos los datos del almacenamiento local mediante la función "split". Esto lo hacemos dentro de la función `init`:

```
// Load saved highscores javascript  
if (localStorage.highscores) {  
    highscores = localStorage.highscores.split(',');  
}
```

Nota que estos datos son siempre separados por una coma. Si almacenarás un arreglo de textos que pueda contener comas, necesitarás un símbolo distinto. Por ejemplo, si quieres almacenar algún texto capturado por parte del usuario, recomiendo usar nuevas líneas (`'\n'`) o tabuladores (`'\t'`), ya que estos son poco probables de ser insertados por el usuario dentro de una caja de texto.

Finalmente, se imprimirán los mejores puntajes en la pantalla. Para ello crearemos una nueva escena donde se mostrarán:

```
// Highscore Scene
highscoresScene = new Scene();

highscoresScene.paint = function (ctx) {
    var i = 0,
        l = 0;

    // Clean canvas
    ctx.fillStyle = '#030';
    ctx.fillRect(0, 0, canvas.width, canvas.height);

    // Draw title
    ctx.fillStyle = '#fff';
    ctx.textAlign = 'center';
    ctx.fillText('HIGH SCORES', 150, 30);

    // Draw high scores
    ctx.textAlign = 'right';
    for (i = 0, l = highscores.length; i < l; i += 1) {
        if (i === posHighscore) {
            ctx.fillText('*' + highscores[i], 180, 40 + i * 10);
        } else {
            ctx.fillText(highscores[i], 180, 40 + i * 10);
        }
    }
};

highscoresScene.act = function () {
    // Load next scene
    if (lastPress === KEY_ENTER) {
        loadScene(gameScene);
        lastPress = null;
    }
};
```

Los altos puntajes son impresos dentro de un ciclo for. En él, comparo si la posición actual es igual a la posición del nuevo alto puntaje, y de ser así, imprimo un asterisco antes del puntaje para conocer cual ha sido el nuevo puntaje dentro de la lista. Habrás notado que al comienzo, declaré la posición con la longitud máxima de los altos puntajes (10), esto para que al comienzo del juego, el asterisco no apareciera en la lista por accidente.

Para llamar a la escena de altos puntajes, he cambiado la llamada a "mainScene" en "gameScene.act" para mostrarlos al final de cada juego. También lo he cambiado la llamada a "gameScene" en "mainScene.act" al comienzo, para mostrar los altos puntajes al comienzo del juego, y saber cuales son los mejores puntajes a vencer.

Con esto, tendremos nuestra lista de mejores puntajes almacenada. Te recuerdo que estos datos son locales en la computadora de cada usuario. Si quieres almacenarlos de forma global para todos los que usen tu juego, esto requiere de un lenguaje servidor como PHP, pero en este curso no nos adentraremos en esa clase de temas.

Para probar el código, lo he implementado al juego de la serpiente. Puedes jugarlo, irte, regresar en unas semanas, y verás que aquí seguirán tus mejores puntuaciones, esperando a que las superes. ¡Diviértete! ¡Y felices códigos!

### Código final:

```
1  /*jslint bitwise:true, es5: true */
2  (function (window, undefined) {
3      'use strict';
4      var KEY_ENTER = 13,
5          KEY_LEFT = 37,
6          KEY_UP = 38,
7          KEY_RIGHT = 39,
8          KEY_DOWN = 40,
9
10     canvas = null,
```

```

11     ctx = null,
12     lastPress = null,
13     pause = false,
14     gameover = false,
15     currentScene = 0,
16     scenes = [],
17     mainScene = null,
18     gameScene = null,
19     highscoresScene = null,
20     body = [],
21     food = null,
22     //var wall = [],
23     highscores = [],
24     posHighscore = 10,
25     dir = 0,
26     score = 0,
27     iBody = new Image(),
28     iFood = new Image(),
29     aEat = new Audio(),
30     aDie = new Audio();
31
32 window.requestAnimationFrame = (function () {
33     return window.requestAnimationFrame ||
34         window.mozRequestAnimationFrame ||
35         window.webkitRequestAnimationFrame ||
36         function (callback) {
37             window.setTimeout(callback, 17);
38         };
39 }());
40
41 document.addEventListener('keydown', function (evt) {
42     if (evt.which >= 37 && evt.which <= 40) {
43         evt.preventDefault();
44     }
45
46     lastPress = evt.which;
47 }, false);
48
49 function Rectangle(x, y, width, height) {
50     this.x = (x === undefined) ? 0 : x;
51     this.y = (y === undefined) ? 0 : y;
52     this.width = (width === undefined) ? 0 : width;
53     this.height = (height === undefined) ? this.width : height;
54 }
55
56 Rectangle.prototype = {
57     constructor: Rectangle,
58
59     intersects: function (rect) {
60         if (rect === undefined) {
61             window.console.warn('Missing parameters on function intersects');
62         } else {
63             return (this.x < rect.x + rect.width &&
64                 this.x + this.width > rect.x &&
65                 this.y < rect.y + rect.height &&
66                 this.y + this.height > rect.y);
67         }
68     },
69
70     fill: function (ctx) {
71         if (ctx === undefined) {
72             window.console.warn('Missing parameters on function fill');
73         } else {
74             ctx.fillRect(this.x, this.y, this.width, this.height);
75         }
76     },
77
78     drawImage: function (ctx, img) {
79         if (img === undefined) {
80             window.console.warn('Missing parameters on function drawImage');
81         } else {
82             if (img.width) {

```

```

83         ctx.drawImage(img, this.x, this.y);
84     } else {
85         ctx.strokeRect(this.x, this.y, this.width, this.height);
86     }
87 }
88 }
89 };
90
91 function Scene() {
92     this.id = scenes.length;
93     scenes.push(this);
94 }
95
96 Scene.prototype = {
97     constructor: Scene,
98     load: function () {},
99     paint: function (ctx) {},
100    act: function () {}
101 };
102
103 function loadScene(scene) {
104     currentScene = scene.id;
105     scenes[currentScene].load();
106 }
107
108 function random(max) {
109     return ~~(Math.random() * max);
110 }
111
112 function addHighscore(score) {
113     posHighscore = 0;
114     while (highscores[posHighscore] > score && posHighscore < highscores.length) {
115         posHighscore += 1;
116     }
117     highscores.splice(posHighscore, 0, score);
118     if (highscores.length > 10) {
119         highscores.length = 10;
120     }
121     localStorage.highscores = highscores.join(',');
122 }
123
124 function repaint() {
125     window.requestAnimationFrame(repaint);
126     if (scenes.length) {
127         scenes[currentScene].paint(ctx);
128     }
129 }
130
131 function run() {
132     setTimeout(run, 50);
133     if (scenes.length) {
134         scenes[currentScene].act();
135     }
136 }
137
138 function init() {
139     // Get canvas and context
140     canvas = document.getElementById('canvas');
141     ctx = canvas.getContext('2d');
142
143     // Load assets
144     iBody.src = 'assets/body.png';
145     iFood.src = 'assets/fruit.png';
146     aEat.src = 'assets/chomp.m4a';
147     aDie.src = 'assets/dies.m4a';
148
149     // Create food
150     food = new Rectangle(80, 80, 10, 10);
151
152     // Create walls
153     //wall.push(new Rectangle(50, 50, 10, 10));
154     //wall.push(new Rectangle(50, 100, 10, 10));

```

```

155     //wall.push(new Rectangle(100, 50, 10, 10));
156     //wall.push(new Rectangle(100, 100, 10, 10));
157
158     // Load saved highscores
159     if (localStorage.highscores) {
160         highscores = localStorage.highscores.split(',');
161     }
162
163     // Start game
164     run();
165     repaint();
166 }
167
168 // Main Scene
169 mainScene = new Scene();
170
171 mainScene.paint = function (ctx) {
172     // Clean canvas
173     ctx.fillStyle = '#030';
174     ctx.fillRect(0, 0, canvas.width, canvas.height);
175
176     // Draw title
177     ctx.fillStyle = '#fff';
178     ctx.textAlign = 'center';
179     ctx.fillText('SNAKE', 150, 60);
180     ctx.fillText('Press Enter', 150, 90);
181 };
182
183 mainScene.act = function () {
184     // Load next scene
185     if (lastPress === KEY_ENTER) {
186         loadScene(highscoresScene);
187         lastPress = null;
188     }
189 };
190
191 // Game Scene
192 gameScene = new Scene();
193
194 gameScene.load = function () {
195     score = 0;
196     dir = 1;
197     body.length = 0;
198     body.push(new Rectangle(40, 40, 10, 10));
199     body.push(new Rectangle(0, 0, 10, 10));
200     body.push(new Rectangle(0, 0, 10, 10));
201     food.x = random(canvas.width / 10 - 1) * 10;
202     food.y = random(canvas.height / 10 - 1) * 10;
203     gameover = false;
204 };
205
206 gameScene.paint = function (ctx) {
207     var i = 0,
208         l = 0;
209
210     // Clean canvas
211     ctx.fillStyle = '#030';
212     ctx.fillRect(0, 0, canvas.width, canvas.height);
213
214     // Draw player
215     ctx.strokeStyle = '#0f0';
216     for (i = 0, l = body.length; i < l; i += 1) {
217         body[i].drawImage(ctx, iBody);
218     }
219
220     // Draw walls
221     //ctx.fillStyle = '#999';
222     //for (i = 0, l = wall.length; i < l; i += 1) {
223     //    wall[i].fill(ctx);
224     //}
225
226     // Draw food

```

```

227     ctx.strokeStyle = '#f00';
228     food.drawImage(ctx, iFood);
229
230     // Draw score
231     ctx.fillStyle = '#fff';
232     ctx.textAlign = 'left';
233     ctx.fillText('Score: ' + score, 0, 10);
234
235     // Debug last key pressed
236     //ctx.fillText('Last Press: '+lastPress,0,20);
237
238     // Draw pause
239     if (pause) {
240         ctx.textAlign = 'center';
241         if (gameover) {
242             ctx.fillText('GAME OVER', 150, 75);
243         } else {
244             ctx.fillText('PAUSE', 150, 75);
245         }
246     }
247 };
248
249 gameScene.act = function () {
250     var i = 0,
251         l = 0;
252
253     if (!pause) {
254         // GameOver Reset
255         if (gameover) {
256             loadScene(highscoresScene);
257         }
258
259         // Move Body
260         for (i = body.length - 1; i > 0; i -= 1) {
261             body[i].x = body[i - 1].x;
262             body[i].y = body[i - 1].y;
263         }
264
265         // Change Direction
266         if (lastPress === KEY_UP && dir !== 2) {
267             dir = 0;
268         }
269         if (lastPress === KEY_RIGHT && dir !== 3) {
270             dir = 1;
271         }
272         if (lastPress === KEY_DOWN && dir !== 0) {
273             dir = 2;
274         }
275         if (lastPress === KEY_LEFT && dir !== 1) {
276             dir = 3;
277         }
278
279         // Move Head
280         if (dir === 0) {
281             body[0].y -= 10;
282         }
283         if (dir === 1) {
284             body[0].x += 10;
285         }
286         if (dir === 2) {
287             body[0].y += 10;
288         }
289         if (dir === 3) {
290             body[0].x -= 10;
291         }
292
293         // Out Screen
294         if (body[0].x > canvas.width - body[0].width) {
295             body[0].x = 0;
296         }
297         if (body[0].y > canvas.height - body[0].height) {
298             body[0].y = 0;

```



```

299     }
300     if (body[0].x < 0) {
301         body[0].x = canvas.width - body[0].width;
302     }
303     if (body[0].y < 0) {
304         body[0].y = canvas.height - body[0].height;
305     }
306
307     // Food Intersects
308     if (body[0].intersects(food)) {
309         body.push(new Rectangle(0, 0, 10, 10));
310         score += 1;
311         food.x = random(canvas.width / 10 - 1) * 10;
312         food.y = random(canvas.height / 10 - 1) * 10;
313         aEat.play();
314     }
315
316     // Wall Intersects
317     //for (i = 0, l = wall.length; i < l; i += 1) {
318     //    if (food.intersects(wall[i])) {
319     //        food.x = random(canvas.width / 10 - 1) * 10;
320     //        food.y = random(canvas.height / 10 - 1) * 10;
321     //    }
322     //
323     //    if (body[0].intersects(wall[i])) {
324     //        gameover = true;
325     //        pause = true;
326     //    }
327     //}
328
329     // Body Intersects
330     for (i = 2, l = body.length; i < l; i += 1) {
331         if (body[0].intersects(body[i])) {
332             gameover = true;
333             pause = true;
334             aDie.play();
335             addHighscore(score);
336         }
337     }
338 }
339 // Pause/Unpause
340 if (lastPress === KEY_ENTER) {
341     pause = !pause;
342     lastPress = null;
343 }
344 };
345
346 // Highscore Scene
347 highscoresScene = new Scene();
348
349 highscoresScene.paint = function (ctx) {
350     var i = 0,
351         l = 0;
352
353     // Clean canvas
354     ctx.fillStyle = '#030';
355     ctx.fillRect(0, 0, canvas.width, canvas.height);
356
357     // Draw title
358     ctx.fillStyle = '#fff';
359     ctx.textAlign = 'center';
360     ctx.fillText('HIGH SCORES', 150, 30);
361
362     // Draw high scores
363     ctx.textAlign = 'right';
364     for (i = 0, l = highscores.length; i < l; i += 1) {
365         if (i === posHighscore) {
366             ctx.fillText('*' + highscores[i], 180, 40 + i * 10);
367         } else {
368             ctx.fillText(highscores[i], 180, 40 + i * 10);
369         }
370     }

```

```
371     };
372
373     highscoresScene.act = function () {
374         // Load next scene
375         if (lastPress === KEY_ENTER) {
376             loadScene(gameScene);
377             lastPress = null;
378         }
379     };
380
381     window.addEventListener('load', init, false);
382 }(window));
```