

Modelos y Simulación para Videojuegos II

ANEXO Programación JavaScript [VERSION PROVISORIA]

1- Programación en JavaScript.

Introducción.

JavaScript (abreviado comúnmente "JS") es un lenguaje de programación interpretado que se utiliza principalmente del lado del cliente (client-side), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. JavaScript se diseñó con una sintaxis similar al C, aunque adopta nombres y convenciones del lenguaje de programación Java. Sin embargo Java y JavaScript no están relacionados y tienen semánticas y propósitos diferentes.



Todos los navegadores modernos interpretan el código JavaScript integrado en las páginas web. Esto lo hace una alternativa sumamente portable. El mismo script puede correr tanto en PC, como en MAC, o mismo en un smart phone. Esta característica es uno de los motivos por los cuales vamos a usar JS para codificar las simulaciones en física para videojuegos.

Además JS es uno de los lenguajes mas difundidos de la actualidad, contando con una comunidad enorme de desarrolladores y un completo ecosistema de librerías, frameworks, foros de discusión, etc.



Top 20 JavaScript Frameworks .

Qué necesitas para trabajar con Javascript?

Para programar en JS necesitamos básicamente un editor de textos y un navegador compatible con JS (por ejemplo el Chrome de Google). Cualquier ordenador mínimamente actual posee de salida todo lo necesario para poder programar en JS.



Aunque el Bloc de Notas es suficiente para empezar, tal vez sea muy útil contar con otros programas que nos ofrecen mejores prestaciones a la hora de escribir las líneas de código. Estos editores avanzados tienen algunas ventajas como que colorean los códigos de nuestros scripts, nos permiten trabajar con varios documentos simultáneamente, tienen ayudas, etc.

Algunos editores gratuitos:

<http://komodoide.com/komodo-edit/>
<http://notepad-plus-plus.org/>
<http://www.ultraedit.com/>

Ejemplo Hello Word

```
1. <!DOCTYPE HTML>
2. <html>
3. <body>
4.
5. <p>Header...</p>
6.
7. <script>
8.   alert('Hello, World!')
9. </script>
10.
11. <p>...Footer</p>
12.
13. </body>
14. </html>
```

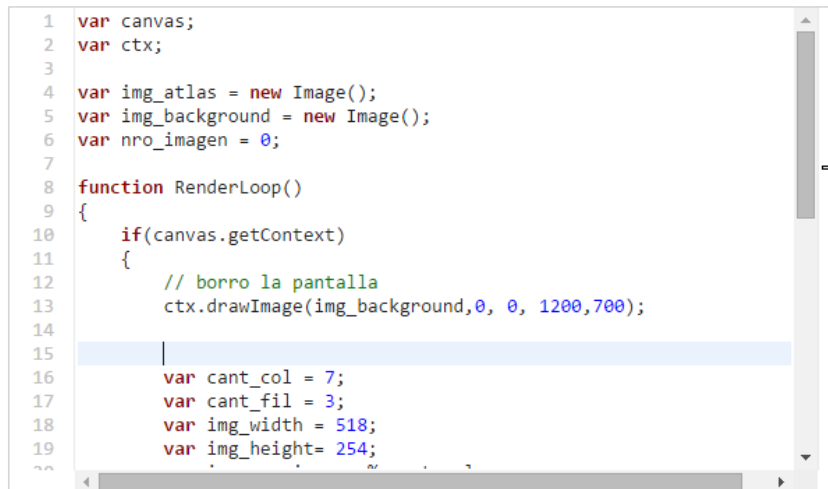
Chequear la Sintaxis.

Como dijimos al principio JS es un lenguaje interpretado, por lo cual, si cometemos un error, este se hará evidente solo en tiempo de ejecución. Eso puede resultar molesto en comparación con otros lenguajes como C++ o C# que son compilados y generan código nativo libre de errores de sintaxis. Sin embargo existen numerosas herramientas y muchas de ellas online, para chequear la sintaxis de los programas escritos en javascript.

Una de las páginas es <http://esprima.org/demo/validate.html>

Podemos pegar nuestro código JS en dicha página para revisar la sintaxis del mismo.

Syntax Validator checks for mistakes and errors

A screenshot of a web-based syntax validator tool. The interface shows a text area with JavaScript code. The code includes variable declarations for 'canvas', 'ctx', 'img_atlas', 'img_background', and 'nro_imagen', followed by a 'RenderLoop' function that checks for context and draws an image. The code is syntactically correct. Below the text area, a green banner displays the message 'Code is syntatically valid.'.

```
1 var canvas;
2 var ctx;
3
4 var img_atlas = new Image();
5 var img_background = new Image();
6 var nro_imagen = 0;
7
8 function RenderLoop()
9 {
10   if(canvas.getContext)
11   {
12     // borro la pantalla
13     ctx.drawImage(img_background,0, 0, 1200,700);
14
15     |
16     var cant_col = 7;
17     var cant_fil = 3;
18     var img_width = 518;
19     var img_height= 254;
```

Code is syntatically valid.

Cuando existen errores de sintaxis, la página informará la línea donde se produce el mismo:

Syntax Validator checks for mistakes and errors

```

1  var canvas;
2  var ctx;
3
4  var img_atlas = new Image();
5  var img_background = new Image();
6  var nro_imagen = 0;
7
8  function RenderLoop()
9  {
10     if(canvas.getContext)
11     {
12         // borro la pantalla
13         ctx.drawImage(img_background,0, 0, 1200,700);
14
15
16         ++3;
17
18         var cant_col = 7;
19         var cant_fil = 3;
20         // ...

```

Invalid code. Total issues: 1

Otra herramienta similar es:

http://www.javascriptlint.com/online_lint.php

Que además informa ciertos warnings y diferencias de estilo y formateo del código.

```

8  function RenderLoop()
9  {
10     if(canvas.getContext)
11     {
12         // borro la pantalla
13         ctx.drawImage(img_background,0, 0, 1200,700);
14
15
16         var cant_col = 7;
17         var cant_fil = 3;
18         var img_width = 518;
19         var img_height= 254;
20         var i = nro_imagen % cant_col;
21         var j = (nro_imagen / cant_col) | 0;
22         var dx = img_width / cant_col;
23         var dy = img_height / cant_fil;
24         ctx.drawImage(img_atlas,i*dx,j*dy , dx,dy ,10, 10,5*dx,5*dy);
25
26         ++nro_imagen;
27         if(nro_imagen==cant_fil*cant_col)
28             nro_imagen = 0;
29

```

===^

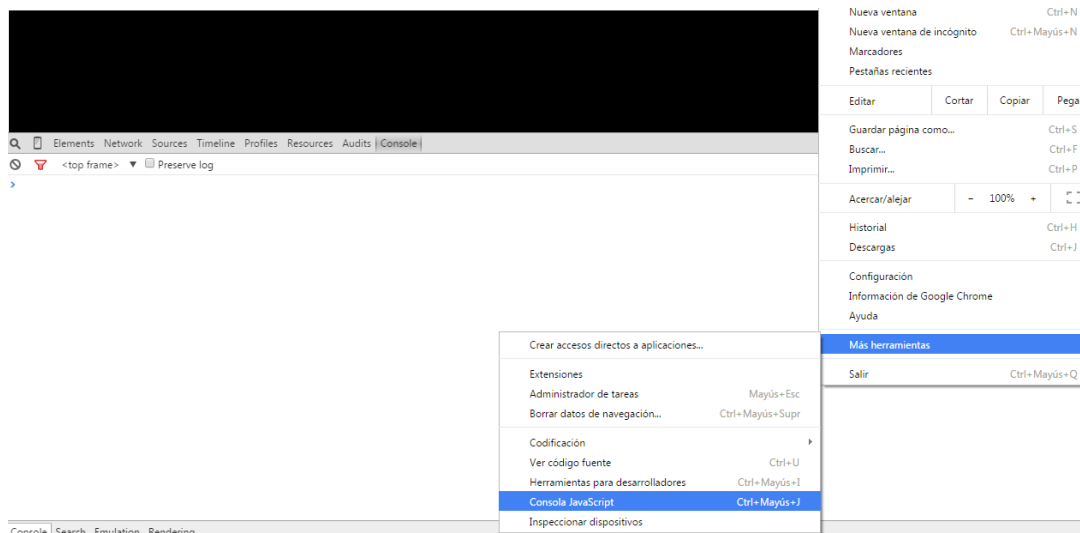
lint warning: block statement without curly braces

29

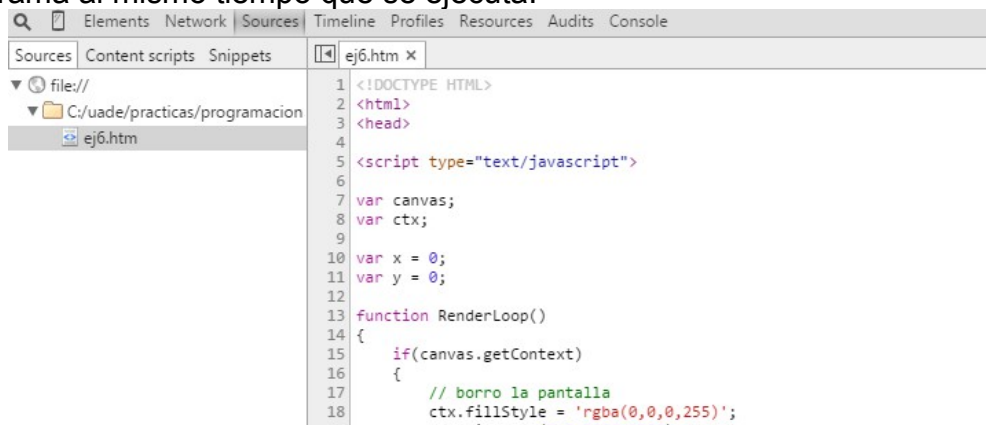
Estas herramientas solo validan errores de sintaxis, y no pueden detectar si escribimos mal un nombre de una función, pues no eso no sería precisamente un error de sintaxis, si no que el problema es que la función no existe. Se supone que la función puede estar en una librería externa, y por lo general no es tan fácil de determinar si se trata de un error de sintaxis o bien la función no está porque no se incluyó la librería.

Una página que detecta más problemas, como usar variables antes de definir las es: <http://codebeautify.org/jsvalidate>

Una vez validado el código para evitar problemas comunes de sintaxis, es inevitable que tenga otro tipo de errores, por ello una de las herramientas que más se usan son los depuradores o debuggers que ya vienen con el navegador. El debugger de Chrome es una herramienta poderosa que cuenta con todas las opciones de los debuggers modernos: posibilidad de inspeccionar variables, poner breakpoints en el código, ejecutar paso a paso, etc. Para llamar al debugger del Chrome podemos ir al menú, luego Más herramientas, y luego la opción Consola JavaScript, o bien presionar Control + Mayúscula + J

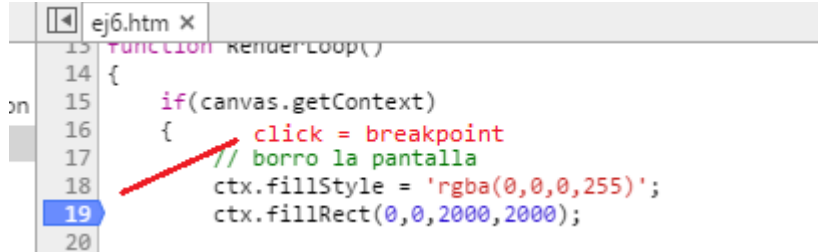


Desde la consola, podemos ir a la solapa Sources, para ver el código del programa al mismo tiempo que se ejecuta:

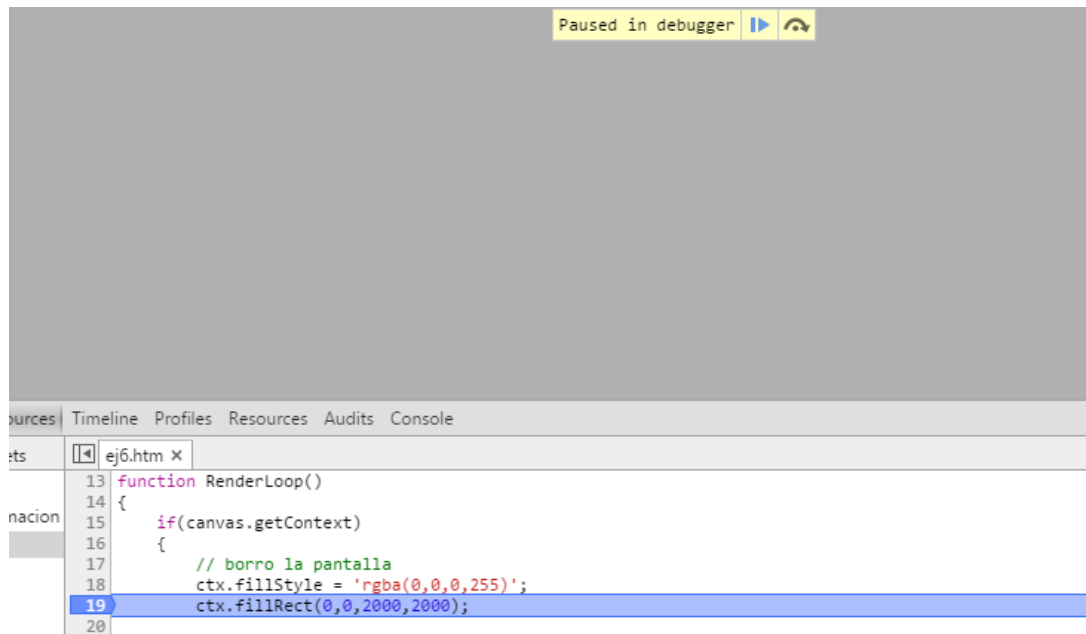


Usualmente vamos a querer poner un breakpoint en el lugar que precisamos, para a partir de ahí debuggear paso a paso.

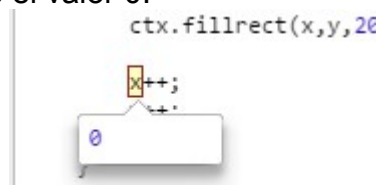
Para poner un breakpoint simplemente hacemos click sobre la línea donde queremos ubicarlo:



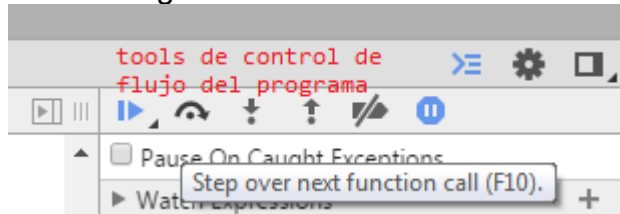
Al ejecutar el programa (con F5 o refresh de la página), la consola detiene la ejecución del código al llegar al breakpoint:



A partir de ahí, podemos inspeccionar el valor o contenido de los objetos, para ello podemos buscar la variable dentro del código, y al posicionar el mouse ya aparece el valor actual de la misma. Por ejemplo la variable x en este momento tiene el valor 0:



Luego se puede avanzar paso a paso o controlar el flujo del programa, se puede ir a las siguientes herramientas :



Supongamos que ponemos la siguiente línea

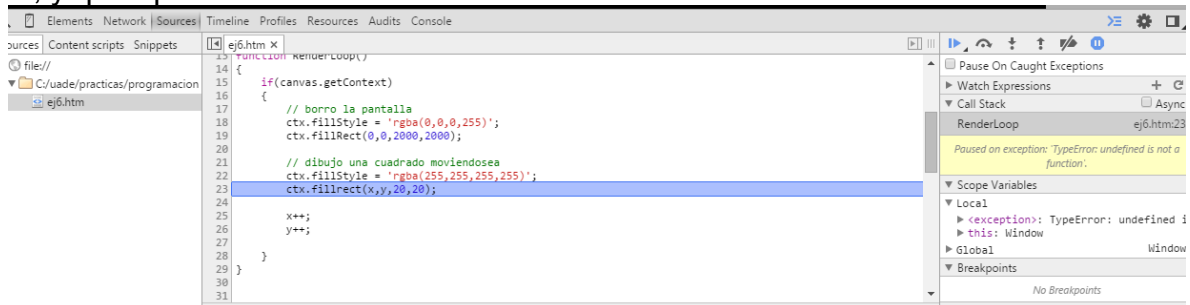
```
ctx.fillRect(x,y,20,20);
```

Donde el error es que la función está mal escrita, y debería ser con la R mayúscula.

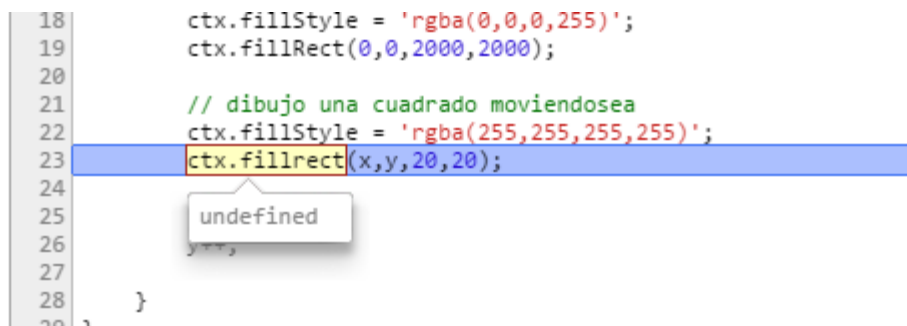
```
ctx.fillRect(x,y,20,20);
```

Al ejecutar el programa desde la consola, usualmente primero se ejecuta el programa, luego se cuelga o no hace lo que debería entonces llamamos al a consola, y luego con F5 o refresh, para que vuelva a ejecutar, esta vez con la consola de javascript activa.

Al saltar el error, se abre la consola y muestra la línea exacta donde ocurrió el error, y que tipo de error es:



Ubicando el mouse sobre la función, indica con un mensaje que esta "indefinida", eso quiere decir que no existe una función que se llama fillrect.



Javascript se escribe en el documento HTML

La programación de JS se realiza dentro del propio documento HTML. Es decir, el código Javascript, en la mayoría de los casos, se mezcla con el propio código HTML para generar la página.

Esto quiere decir que debemos aprender a mezclar los dos lenguajes de programación y rápidamente veremos que, para que estos dos lenguajes puedan convivir sin problemas entre ellos, se han de incluir unos delimitadores que separan las etiquetas HTML de las instrucciones Javascript. Estos delimitadores son las etiquetas `<SCRIPT>` y `</SCRIPT>`. Todo el código Javascript que pongamos en la página ha de ser introducido entre estas dos etiquetas.

Incluir ficheros externos de Javascript

Una forma incluir scripts en páginas web, es incluir archivos externos donde se pueden colocar muchas funciones que se utilicen en la página. Los ficheros suelen tener extensión `.js` y se incluyen de esta manera.

```
<SCRIPT type="text/javascript" src="archivo_externo.js"></SCRIPT>
```

El archivo que incluimos (en este caso `archivo_externo.js`) debe contener tan solo sentencias Javascript. No debemos incluir código HTML de ningún tipo, ni tan siquiera las etiquetas `</SCRIPT>` y `</SCRIPT>`.

Comentarios en el código

Un comentario es una parte de código que no es interpretada por el navegador y cuya utilidad radica en facilitar la lectura al programador. El programador, a medida que desarrolla el script, va dejando frases o palabras sueltas, llamadas comentarios, que le ayudan a él o a cualquier otro a leer mas fácilmente el script a la hora de modificarlo o depurarlo. Existen dos tipos de comentarios en el lenguaje. Uno de ellos, la doble barra, sirve para comentar una línea de código. El otro comentario lo podemos utilizar para comentar varias líneas y se indica con los signos `/*` para empezar el comentario y `*/` para terminarlo. Veamos unos ejemplos.

```
1. <SCRIPT>
2. //Este es un comentario de una línea
3. /*Este comentario se puede extender
4. por varias líneas.
5. Las que quieras*/
6. </SCRIPT>
```

Mayúsculas y minúsculas

En Javascript se han de respetar las mayúsculas y las minúsculas. Si nos equivocamos al utilizarlas el navegador responderá con un mensaje de error, ya sea de sintaxis o de referencia indefinida.

Por poner un ejemplo, no es lo mismo la función `alert()` que la función `Alert()`. La primera muestra un texto en una caja de diálogo y la segunda (con la primera A mayúscula) simplemente no existe, a no ser que la definamos nosotros. Otro claro ejemplo lo veremos cuando tratemos con variables, puesto que los nombres que damos a las variables también son sensibles a las mayúsculas y minúsculas.

JavaScript Output.

JS no tiene funciones para imprimir la salida, así como C++ o C# cuenta con la función `printf`. Si embargo para poder imprimir algún resultado, se puede usar `document.write()` que directamente "imprime" el resultado en la página web. Para pasar a la siguiente línea se puede usar el HTML tag `
`

Ejemplo. Imprimir los primeros 10 números enteros positivos.

```
1. <!DOCTYPE html>
2. <html>
3. <body>
4. <h1>Ejercicio 1</h1>
5. <script>
6.   for(var i = 0; i<10; ++i)
7.     document.write(i + "<br>");
8. </script>
9. </body>
10. </html>
```

Declaración de variables en Javascript

Declarar variables consiste en definir y de paso informar al sistema de que vas a utilizar una variable. Es una costumbre habitual en los lenguajes de programación el definir las variables que se van a usar en los programas y para ello, se siguen unas reglas estrictas. Pero Javascript se salta muchas reglas por ser un lenguaje un tanto libre a la hora de programar y uno de los casos en los que otorga un poco de libertad es a la hora de declarar las variables, ya que no estamos obligados a hacerlo, al contrario de lo que pasa en la mayoría de los lenguajes de programación.

Javascript cuenta con la palabra `"var"` que utilizaremos cuando queramos declarar una o varias variables. Como es lógico, se utiliza esa palabra para definir la variable antes de utilizarla.

Aunque Javascript no nos obligue a declarar explícitamente las variables, es aconsejable declararlas antes de utilizarlas y veremos en adelante que se trata también de una buena costumbre.

```
var operando1;
var operando2;
```

También se puede asignar un valor a la variable cuando se está declarando

```
var operando1 = 23;
var operando2 = 33;
```

También se permite declarar varias variables en la misma línea, siempre que se separen por comas.

```
var operando1 = 23 , operando2 = 33;
```

Ambito de las variables en Javascript.

El ámbito de las variables es uno de los conceptos más importantes que deberemos conocer cuando trabajamos con variables, no sólo en Javascript, sino en la mayoría de los lenguajes de programación.

Se le llama ámbito de las variables al lugar donde estas están disponibles. Por lo general, cuando declaramos una variable hacemos que esté disponible en el lugar donde se ha declarado, esto ocurre en todos los lenguajes de programación y como Javascript se define dentro de una página web, las variables que declaremos en la página estarán accesibles dentro de ella.

En Javascript no podremos acceder a variables que hayan sido definidas en otra página. Por tanto, la propia página donde se define es el ámbito más habitual de una variable y le llamaremos a este tipo de variables globales a la página. Veremos también se pueden hacer variables con ámbitos distintos del global, es decir, variables que declaremos y tendrán validez en lugares más acotados.

Variables globales

Como hemos dicho, las variables globales son las que están declaradas en el ámbito más amplio posible, que en Javascript es una página web. Para declarar una variable global a la página simplemente lo haremos en un script, con la palabra var.

```
1. <SCRIPT>
2. var elapsedTime = 0;
3. </SCRIPT>
```

Las variables globales son accesibles desde cualquier lugar de la página, es decir, desde el script donde se han declarado y todos los demás scripts de la página, incluidos los manejadores de eventos, como el onclick, que veremos más adelante.

Variables locales

También podremos declarar variables en lugares más acotados, como por ejemplo una función. A estas variables les llamaremos locales. Cuando se declaren variables locales sólo podremos acceder a ellas dentro del lugar donde se ha declarado, es decir, si la habíamos declarado en una función solo podremos acceder a ella cuando estemos en esa función.

Las variables pueden ser locales a una función, pero también pueden ser locales a otros ámbitos, como por ejemplo un bucle. En general, son ámbitos locales cualquier lugar acotado por llaves.

```
1. <SCRIPT>
2. function miFuncion (){
3.     var variableLocal = 0;
4. }
5. </SCRIPT>
```

En el script anterior hemos declarado una variable dentro de una función, por lo que esa variable sólo tendrá validez dentro de la misma. Se pueden ver cómo se utilizan las llaves para acotar el lugar donde está definida esa función o su ámbito.

No hay problema en declarar una variable local con el mismo nombre que una global, en este caso la variable global será visible desde toda la página, excepto en el ámbito donde está declarada la variable local ya que en este sitio ese nombre de variable está ocupado por la local y es ella quien tiene validez. En resumen, la variable que tendrá validez en cualquier sitio de la página es la global. Menos en el ámbito donde está declarada la variable local, que será ella quien tenga validez.

```
1. <SCRIPT>
2. var numero = 2
3. function miFuncion (){
4.     var numero = 19
5.     document.write(numero) //imprime 19
6. }
7. document.write(numero) //imprime 2
8. </SCRIPT>
```

OJO:

Cuando utilizamos var estamos haciendo que la variable que estamos declarando sea local al ámbito donde se declara. Por otro lado, si NO utilizamos la palabra var para declarar una variable, ésta será global a toda la página, sea cual sea el ámbito en el que haya sido declarada.

Tipos de Datos en JS.

Números

Para empezar tenemos el tipo numérico, para guardar números como 9 o 23.6

```
var pi = 3.1415;
```

En JS sólo existe un tipo de datos numérico, al contrario que ocurre en la mayoría de los lenguajes más conocidos. Todos los números son por tanto del tipo numérico, independientemente de la precisión que tengan o si son números reales

o enteros. Los números enteros son números que no tienen coma, como 3 o 339. Los números reales son números fraccionarios, como 2.69 o 0.25, que también se pueden escribir en notación científica, por ejemplo 2.482e12.

A veces suele resultar útil escribir números en base hexadecimal. La base 16 o sistema hexadecimal, es el sistema de numeración que utiliza 16 dígitos, los comprendidos entre el 0 y el 9 y las letras de la A a la F, para los dígitos que faltan. Para escribir un número en hexadecimal debemos escribirlo precedido de un cero y una equis, por ejemplo 0x3EF.

El JS no es un lenguaje fuertemente tipado. Eso quiere decir que no es muy estricto en cuanto a las reglas para determinar si una variable es un número entero, double, float etc. Eso muchas veces puede generar problemas cuando lo que se quiere es tomar un número entero. El JS tampoco tiene operadores de "cast" como el c++. La forma habitual de castear una expresión a entero es usar el operador | (logic or operator).

Por ejemplo, en la instrucción:

```
var p2 = (p/2) | 0;
```

La variable p2 tendrá la mitad del valor de p, pero truncado a un entero. Si p = 11, luego $p2 = 11/2 = [5,5] = 5$

Cadenas o Strings

El tipo cadena de carácter guarda un texto. Siempre que escribamos una cadena de caracteres debemos utilizar las comillas (").

```
var nombre = "Peppa Pig";
```

Todo lo que se coloca entre comillas es tratado como una cadena de caracteres independientemente de lo que coloquemos en el interior de las comillas. Por ejemplo, en una variable de texto podemos guardar números y en ese caso tenemos que tener en cuenta que las variables de tipo texto y las numéricas no son la misma cosa y mientras que las de numéricas nos sirven para hacer cálculos matemáticos las de texto no.

Muchas veces se usan las comillas simples,

```
var nombre = 'Peppa Pig';
```

Boleanos

El tipo boolean, sirve para guardar un SI o un NO, o dicho de otro modo, un verdadero o un falso. Se utiliza para realizar operaciones lógicas, generalmente para realizar acciones si el contenido de una variable es verdadero o falso.

Los dos valores que pueden tener las variables booleanas son true o false.

```
var hit = false;
var seguir = true;
```

Por último sería relevante señalar aquí que nuestras variables pueden contener cosas más complicadas, como podría ser un objeto, una función, o vacío (null).

Se dice que JS es un lenguaje débilmente tipado, porque a diferencia de C++, o C# permite hacer algo así:

```
var nombre = "Peppa Pig";
nombre = 100;
```

Operadores Javascript

Operadores aritméticos

Son los utilizados para la realización de operaciones matemáticas simples como la suma, resta o multiplicación. En JS son los siguientes:

- + Suma de dos valores
- Resta de dos valores, también puede utilizarse para cambiar el signo de un número si lo utilizamos con un solo operando -23
- * Multiplicación de dos valores
- / División de dos valores
- % El resto de la división de dos números (3%2 devolvería 1, el resto de dividir 3 entre 2)
- ++ Incremento en una unidad, se utiliza con un solo operando
- Decremento en una unidad, utilizado con un solo operando

Ejemplo

```
1. //introduzco un 128 en la variable precio
2. precio = 128;
3.
4. //otra asignacion, luego veremos operadores de asignacion
5. unidades = 10;
6.
7. //multiplico precio por unidades, obtengo el valor factura
8. factura = precio * unidades;
9.
10. //obtengo el resto de dividir la variable factura por 3
11. resto = factura % 3;
12.
13. //incrementa en una unidad el precio (ahora vale 129)
14. precio++;
```

Operadores de asignación

Sirven para asignar valores a las variables, ya hemos utilizado en ejemplos anteriores el operador de asignación =, pero hay otros operadores de este tipo, que provienen del lenguaje C y que muchos ya conocerán.

= Asignación. Asigna la parte de la derecha del igual a la parte de la izquierda. A la derecha se colocan los valores finales y a la izquierda generalmente se coloca una variable donde queremos guardar el dato.

+= Asignación con suma. Realiza la suma de la parte de la derecha con la de la izquierda y guarda el resultado en la parte de la izquierda.

-= Asignación con resta

*= Asignación de la multiplicación

/= Asignación de la división

%= Se obtiene el resto y se asigna

Ejemplo:

```
1. //asigna un 7000 a la variable ahorros
2. ahorros = 7000;
3.
4. //incrementa en 3500 la variable ahorros, ahora vale 10500
5. ahorros += 3500;
6.
7. //divide entre 2 mis ahorros, ahora quedan 5250
8. ahorros /= 2
```

Operadores de cadenas

Las cadenas de caracteres, o variables de texto, también tienen sus propios operadores para realizar acciones típicas sobre cadenas. Aunque JS sólo tiene un operador para cadenas se pueden realizar otras acciones con una serie de funciones predefinidas en el lenguaje que veremos más adelante.

+ Concatena dos cadenas, pega la segunda cadena a continuación de la primera.

Ejemplo

```
1. var s1 = "hola";
2. var s2 = "mundo";
3. var s = s1 + s2; // s vale "holamundo"
```

Un detalle importante que se puede ver en este caso es que el operador + sirve para dos usos distintos, si sus operandos son números los suma, pero si se trata de cadenas las concatena.

Un caso que resultaría confuso es el uso del operador + cuando se realiza la operación con operadores texto y numéricos entremezclados. En este caso javascript asume que se desea realizar una concatenación y trata a los dos operandos como si de cadenas de caracteres se trataran, incluso si la cadena de texto que tenemos fuese un número. Esto lo veremos más fácilmente con el siguiente ejemplo.

```

1. var r = 100;
2. var g = 200;
3. var b = 0;
4. // color vale "rgba(100,200,0,255)"
5. var color = "rgba(" + r + "," + g + "," + b + ", 255)";

```

Operadores lógicos

Estos operadores sirven para realizar operaciones lógicas, que son aquellas que dan como resultado un verdadero o un falso, y se utilizan para tomar decisiones en nuestros scripts. En vez de trabajar con números, para realizar este tipo de operaciones se utilizan operandos booleanos, que conocimos anteriormente, que son el verdadero (true) y el falso (false). Los operadores lógicos relacionan los operandos booleanos para dar como resultado otro operando booleano, tal como podemos ver en el siguiente ejemplo.

```

1. if (x==2 && y!=3){
2.     //la variable x vale 2 y la variable y es distinta de tres
3. }

```

Operadores condicionales

Sirven para realizar expresiones condicionales todo lo complejas que deseemos. Estas expresiones se utilizan para tomar decisiones en función de la comparación de varios elementos, por ejemplo si un número es mayor que otro o si son iguales.

Los operadores condicionales se utilizan en las expresiones condicionales para tomar decisiones. Como estas expresiones condicionales serán objeto de estudio más adelante será mejor describir los operadores condicionales más adelante. De todos modos aquí podemos ver la tabla de operadores condicionales.

==	Comprueba si dos valores son iguales
!=	Comprueba si dos valores son distintos
>	Mayor que, devuelve true si el primer operando es mayor que el segundo
<	Menor que, es true cuando el elemento de la izquierda es menor que el de la derecha
>=	Mayor igual
<=	Menor igual

Estructuras de Control.

Las estructuras de control en JS son similares al lenguaje C,

IF

Ejemplo simple.


```

1.  if (expression) {
2.      //acciones a realizar en caso positivo
3.      //...
4.  }

```

Ejemplo if..else.

```

1.  if (expression) {
2.      //acciones a realizar en caso positivo
3.      //...
4.  } else {
5.      //acciones a realizar en caso negativo
6.      //...
7.  }

```

Ejemplo if anidados.

```

1.  var numero1=23;
2.  var numero2=63;
3.  if (numero1 == numero2){
4.      document.write("Los dos números son iguales")
5.  }else{
6.      if (numero1 > numero2) {
7.          document.write("El primer número es mayor que el segundo")
8.      }else{
9.          document.write("El primer número es menor que el segundo")
10.     }
11. }

```

Operador IF

Hay un operador que no hemos visto todavía y es una forma más esquemática de realizar algunos IF sencillos. Proviene del lenguaje C, donde se escriben muy pocas líneas de código y donde cuanto menos escribamos más elegantes seremos. Este operador es un claro ejemplo de ahorro de líneas y caracteres al escribir los scripts. Lo veremos rápidamente, pues la única razón por la que lo incluyo es para que sepas que existe y si lo encuentras en alguna ocasión por ahí sepas identificarlo y cómo funciona.

Un ejemplo de uso del operador IF se puede ver a continuación.

Variable = (condición) ? valor1 : valor2

Este ejemplo no sólo realiza una comparación de valores, además asigna un valor a una variable. Lo que hace es evaluar la condición (colocada entre paréntesis) y si es positiva asigna el valor1 a la variable y en caso contrario le asigna el valor2.

Ejemplo:

```

1.  momento = (hora_actual < 12) ?
2.      "Antes del mediodía" :
3.      "Después del mediodía";

```

SWITCH

```
1. switch (expresión) {
2.     case valor1:
3.         Sentencias a ejecutar si la expresión tiene como valor a valor1
4.         break
5.     case valor2:
6.         Sentencias a ejecutar si la expresión tiene como valor a valor2
7.         break
8.     case valor3:
9.         Sentencias a ejecutar si la expresión tiene como valor a valor3
10.        break
11.    default:
12.        Sentencias a ejecutar si el valor no es ninguno de los anteriores
13. }
```

FOR

```
1. for (inicialización; condición; actualización) {
2.     //sentencias a ejecutar en cada iteración
3. }
```

Un ejemplo de utilización de este bucle lo podemos ver a continuación, donde se imprimirán los números del 0 al 10.

```
1. var i;
2. for (i=0;i<=10;i++) {
3.     document.write(i);
4.     document.write("<br>");
5. }
```

En este caso se inicializa la variable *i* a 0. Como condición para realizar una iteración, se tiene que cumplir que la variable *i* sea menor o igual que 10. Como actualización se incrementará en 1 la variable *i*.

WHILE

```
while (condición){
    //sentencias a ejecutar
}
```

En este ejemplo vamos a declarar una variable e inicializarla a 0. Luego iremos sumando a esa variable un número aleatorio del 1 al 100 hasta que sumemos 1.000 o más, imprimiendo el valor de la variable suma después de cada operación. Será necesario utilizar el bucle WHILE porque no sabemos exactamente el número de iteraciones que tendremos que realizar (dependerá de los valores aleatorios que se vayan obteniendo).

```
1. var suma = 0;
2. while (suma < 1000){
3.     suma += parseInt(Math.random() * 100);
4.     document.write (suma + "<br>");
5. }
```

Break

Se detiene un bucle utilizando la palabra `break`. Detener un bucle significa salirse de él y dejarlo todo como está para continuar con el flujo del programa inmediatamente después del bucle.

```
1. for (i=0;i<10;i++){
2.     document.write (i);
3.     escribe = prompt("dime si continuo preguntando...", "si");
4.     if (escribe == "no")
5.         break
6. }
```

Este ejemplo escribe los números del 0 al 9 y en cada iteración del bucle pregunta al usuario si desea continuar. Si el usuario dice cualquier cosa continua, excepto cuando dice "no", situación en la cual se sale del bucle y deja la cuenta por donde se había quedado.

Continue

Sirve para volver al principio del bucle en cualquier momento, sin ejecutar las líneas que haya por debajo de la palabra `continue`.

```
1. var i=0;
2. while (i<7){
3.     incrementar = prompt("La cuenta está en " + i + ", dime si incremento", "si")
4.     if (incrementar == "no")
5.         continue
6.     i++;
7. }
```

Este ejemplo, en condiciones normales contaría hasta desde $i=0$ hasta $i=7$, pero cada vez que se ejecuta el bucle pregunta al usuario si desea incrementar la variable o no. Si introduce "no" se ejecuta la sentencia `continue`, con lo que se vuelve al principio del bucle sin llegar a incrementar en 1 la variable i , ya que se ignorarían las sentencias que haya por debajo del `continue`.

Funciones en JavaScript.

Para definir una función en JS primero se escribe la palabra `function`, reservada para este uso. Seguidamente se escribe el nombre de la función, que como los nombres de variables puede tener números, letras y algún carácter adicional como en guión bajo. A continuación se colocan entre llaves las distintas instrucciones de la función. Las llaves en el caso de las funciones no son opcionales, además es útil colocarlas siempre como se ve en el ejemplo, para que se reconozca fácilmente la estructura de instrucciones que engloba la función.

```
1. function mifuncion(){
2.     instrucciones de la función
3.     ...
4. }
```

Para ejecutar una función la tenemos que invocar en cualquier parte de la página. Con eso conseguiremos que se ejecuten todas las instrucciones que tiene la función entre las dos llaves.

Para ejecutar la función utilizamos su nombre seguido de los paréntesis. Por ejemplo, así llamaríamos a la función `miFuncion` que acabamos de crear. Los parámetros de la función se declaran separados por coma, y solo con el nombre de la variable que los va a recibir, como JS no es tipado, no se coloca el tipo de dato del parámetro.

```
1. function escribirBienvenida(nombre,colorTexto){
2.     document.write("<FONT color=" + colorTexto + ">")
3.     document.write("<H1>Hola " + nombre + "</H1>")
4.     document.write("</FONT>");
5. }
```

Para llamar a la función:

```
1. var miNombre = "Pepe";
2. var miColor = "red";
3. escribirBienvenida(miNombre,miColor);
```

Los parámetros se pasan por valor!!!

Tenemos que saber que los parámetros de las funciones se pasan por valor. Esto quiere decir que estamos pasando valores y no variables. En la práctica, aunque modifiquemos un parámetro en una función, la variable original que habíamos pasado no cambiará su valor. Se puede ver fácilmente con un ejemplo.

```
1. function pasoPorValor(miParametro){
2.     miParametro = 32;
3.     document.write("he cambiado el valor a 32");
4. }
5. var miVariable = 5;
6. pasoPorValor(miVariable);
7. document.write ("el valor de la variable es: " + miVariable);
```

En el ejemplo tenemos una función que recibe un parámetro y que modifica el valor del parámetro asignándole el valor 32. También tenemos una variable, que inicializamos a 5 y posteriormente llamamos a la función pasándole esta variable como parámetro. Como dentro de la función modificamos el valor del parámetro podría pasar que la variable original cambiase de valor, pero como los parámetros no modifican el valor original de las variables, ésta no cambia de valor.

De este modo, una vez ejecutada la función, al imprimir en pantalla el valor de `miVariable` se imprimirá el número 5, que es el valor original de la variable, en lugar de 32 que era el valor con el que habíamos actualizado el parámetro.

En Javascript sólo se pueden pasar las variables por valor.

Las funciones pueden devolver valores, a través de la sentencia `return`, De modo que, al invocar una función, se podrá realizar acciones y ofrecer un valor como salida.

Veamos un ejemplo de función que calcula el promedio de dos números. La función recibirá los dos números y retornará el valor promedio

```
1. function media(valor1,valor2){  
2.     var resultado;  
3.     resultado = (valor1 + valor2) / 2;  
4.     return resultado;  
5. }
```

Para especificar el valor que retornará la función se utiliza la palabra return seguida de el valor que se desea devolver. En este caso se devuelve el contenido de la variable resultado, que contiene la media calculada de los dos números.

Dentro de las funciones podemos declarar variables. Las variables declaradas en una función son locales a esa función, es decir, sólo tendrán validez durante la ejecución de la función.

Creación de Arrays javascript

El navegador tiene "hardcodeado" una función para crear arrays. Esta es la sentencia para crear un objeto array:

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

El array se crea sin ningún contenido, es decir, no tendrá ninguna casilla o compartimiento creado. También podemos crear el array Javascript especificando el número de elementos que va a tener.

```
var miArray = new Array(10);
```

En este caso indicamos que el array va a tener 10 posiciones.

Es importante que nos fijemos que la palabra Array en código Javascript se escribe con la primera letra en mayúscula. Como en Javascript las mayúsculas y minúsculas si que importan, si lo escribimos en minúscula no funcionará.

Tanto se indique o no el número de casillas del array javascript, podemos introducir en el array cualquier dato. Si la casilla está creada se introduce simplemente y si la casilla no estaba creada se crea y luego se introduce el dato, con lo que el resultado final es el mismo. Esta creación de casillas es dinámica y se produce al mismo tiempo que los scripts se ejecutan. Veamos a continuación cómo introducir valores en nuestros arrays.

```
1. miArray[0] = 290;  
2. miArray[1] = 97;  
3. miArray[2] = 127;
```

Se introducen indicando entre corchetes el índice de la posición donde queríamos guardar el dato. En este caso introducimos 290 en la posición 0, 97 en la posición 1 y 127 en la 2.

Los arrays en Javascript empiezan siempre en la posición 0, así que un array que tenga por ejemplo 10 posiciones, tendrá casillas de la 0 a la 9. Para recoger datos de un array lo hacemos igual: poniendo entre corchetes el índice de la

posición a la que queremos acceder. Veamos cómo se imprimiría en la pantalla el contenido de un array.

```
1. var miArray = new Array(3);
2. miArray[0] = 155;
3. miArray[1] = 4;
4. miArray[2] = 499;
5.
6. for (i=0;i<3;i++){
7.     document.write("Posición " + i + " del array: " + miArray[i]);
8.     document.write("<br>");
9. }
```

Tipos de datos en los arrays

En las casillas de los arrays podemos guardar datos de cualquier tipo. Podemos ver un array donde introducimos datos de tipo carácter.

```
1. miArray[0] = "Hola";
2. miArray[1] = "a";
3. miArray[2] = "todos";
```

Incluso, en Javascript podemos guardar distintos tipos de datos en los elementos de un mismo array. Es decir, podemos introducir números en algunos elementos, textos en otros, booleanos o cualquier otra cosa que deseemos.

```
1. miArray[0] = "Peppa Pig"
2. miArray[1] = 1275;
3. miArray[1] = 0.78;
4. miArray[2] = true;
```

Declaración e inicialización resumida de Arrays

En Javascript tenemos a nuestra disposición una manera resumida de declarar un array y cargar valores en un mismo paso.

Ejemplo:

```
var arrayRapido = [12,45,"array inicializado en su declaración"];
```

2- HTML Canvas.



El canvas es un elemento de HTML usado para dibujar gráficos “on the fly”, generalmente a través de JS. El elemento canvas es un sólo un contenedor donde se van a dibujar los gráficos. Toda la lógica de renderizado va en un script de JS que accede al elemento canvas para dibujar.

Ejemplo: Crear un canvas de 200 x 100 de tamaño.

```
<canvas id="myCanvas" width="200" height="100" > </canvas>
```

Dibujar con el canvas.

Para dibujar dentro del canvas es preciso acceder al mismo a través de lo que se llama drawing context, ese objeto, nos va a permitir llamar interactuar con el canvas. Todas las funciones gráficas van a tomar como parámetro el drawing context.

Dibujar un rectángulo:

```
1. var c=document.getElementById("myCanvas");  
2. var ctx = c.getContext("2d");  
3. ctx.fillRect(20,20,150,100);  
4. ctx.stroke();
```



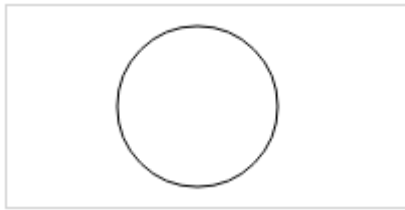
Dibujar una línea:

```
1. var c = document.getElementById("myCanvas");  
2. var ctx = c.getContext("2d");  
3. ctx.moveTo(0,0);  
4. ctx.lineTo(200,100);  
5. ctx.stroke();
```



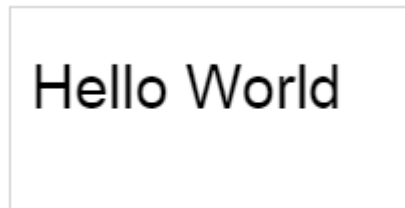
Dibujar un círculo:

```
1. var c = document.getElementById("myCanvas");  
2. var ctx = c.getContext("2d");  
3. ctx.beginPath();  
4. ctx.arc(95,50,40,0,2*Math.PI);  
5. ctx.stroke();
```



Dibujar un Texto:

```
1. var c = document.getElementById("myCanvas");  
2. var ctx = c.getContext("2d");  
3. ctx.font = "30px Arial";  
4. ctx.fillText("Hello World",10,50);
```



Resumen de las principales propiedades del objeto context.

Muchas propiedades como el color y espesor de línea, el color de relleno, etc, son establecidas directamente llamando a alguna propiedad del objeto context, como vimos en todos los ejemplos anteriores

ctx.propiedad = XX;

A continuación un resumen de las principales que vamos a utilizar durante el curso.

Para una lista exhaustiva consultar HTML Canvas Reference :

http://www.w3schools.com/tags/ref_canvas.asp

Propiedad	Descripción
<u>fillStyle</u>	Sets or returns the color, gradient, or pattern used to fill the drawing
<u>strokeStyle</u>	Sets or returns the color, gradient, or pattern used for strokes
<u>lineWidth</u>	Sets or returns the current line width
<u>fillRect()</u>	Draws a "filled" rectangle
<u>strokeRect()</u>	Draws a rectangle (no fill)
<u>clearRect()</u>	Clears the specified pixels within a given rectangle
<u>fill()</u>	Fills the current drawing (path)
<u>stroke()</u>	Actually draws the path you have defined
<u>beginPath()</u>	Begins a path, or resets the current path
<u>moveTo()</u>	Moves the path to the specified point in the canvas, without creating a line
<u>closePath()</u>	Creates a path from the current point back to the starting point
<u>lineTo()</u>	Adds a new point and creates a line from that point to the last specified point in the canvas
<u>arc()</u>	Creates an arc/curve (used to create circles, or parts of circles)
<u>arcTo()</u>	Creates an arc/curve between two tangents
<u>font</u>	Sets or returns the current font properties for text content
<u>textAlign</u>	Sets or returns the current alignment for text content
<u>fillText()</u>	Draws "filled" text on the canvas
<u>strokeText()</u>	Draws text on the canvas (no fill)
<u>measureText()</u>	Returns an object that contains the width of the specified text

3- Animaciones en Canvas.

Para dibujar una animación adentro de un Canvas necesitamos que el contenido se redibuje constantemente, a intervalos regulares, de tal forma que genere la sensación de que se está moviendo. Para lograr eso se usa la función de JS setInterval, a la cual se le pasa como parámetro la función queremos que se llame y cada cuantos milisegundos:

```
setInterval(RenderLoop, 1000);
```

En este ejemplo, la función RenderLoop, que tiene que estar definida anteriormente, se va a llamar continuamente cada 1000 milisegundos. Es decir una vez por segundo.

Dentro de la función RenderLoop tendremos que repintar el contenido del canvas para lograr la animación.

Ejemplo. dibujo un cuadrado blanco moviéndose sobre un fondo negro:

```
1. <!DOCTYPE HTML>
2. <html>
3. <head>
4. <script type="text/javascript">
5.   var canvas;
6.   var ctx;
7.   var x = 0;
8.   var y = 0;
9.
10.  function RenderLoop()
11.  {
12.    if(canvas.getContext)
13.    {
14.      // borro la pantalla
15.      ctx.fillStyle = 'rgba(0,0,0,255)';
16.      ctx.fillRect(0,0,2000,2000);
17.
18.      // dibujo una cuadrado moviendosea
19.      ctx.fillStyle = 'rgba(255,255,255,255)';
20.      ctx.fillRect(x,y,20,20);
21.
22.      x++;
23.      y++;
24.
25.    }
26.  }
27.
28.  function main()
29.  {
30.    canvas = document.getElementById('mycanvas');
31.    ctx = canvas.getContext('2d');
32.    setInterval(RenderLoop, 1000 / 60);
33.  }
34.
35. </script>
36. </head>
37. <body onload="main();">
38.   <canvas id="mycanvas" width="1100" height="650"></canvas>
39. </body>
40. </html>
```

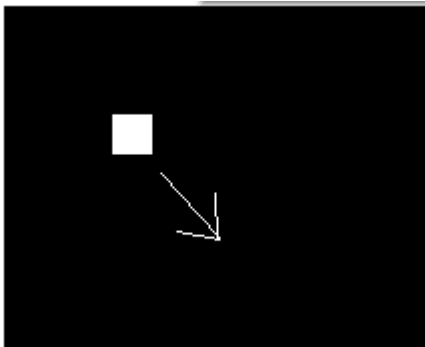
La estructura de los programas con animaciones son un poco más complicadas y hay algunas cuestiones a tener en cuenta. Al cargar el HTML se

llama a la función `main()` que se encarga de acceder a canvas y obtener el context para dibujar. Ese context lo guarda como una variable global así es accesible desde todas partes y nos va a permitir interactuar con el canvas. Luego llama a la función `setInterval` para indicar que `RenderLoop` se va a llamar 60 veces por segundo. (1000/60)

Es posible que las primeras veces que se llame `RenderLoop` el context todavía no esté inicializado, así que se suele agregar esta línea para asegurarse de no dibujar nada hasta que el canvas este listo:

```
if (canvas.getContext)
{
    ... drawing
}
```

Finalmente la función `RenderLoop` tiene el código para dibujar un cuadrado en una cierta posición que se va actualizando en cada frame. Es decir 60 veces por segundo se esta llamando este código que no solo dibuja el cuadrado si no que también actualiza la posición x,y donde se va a dibujar. Este produce que en cada frame el cuadrado va a aparecer en un lugar diferente, dando la sensación de que se está moviendo.



Dibujar Imágenes.

Para dibujar una imagen en el Canvas se puede usar la función `drawImage` de JS. Antes de llamar a la función tenemos que cargar la imagen a un objeto de memoria. Para ello se usa la clase de JS `Image` que encapsula todo lo relativo a imágenes 2d.

La siguiente línea crea un objeto imagen:

```
var img_ball = new Image();
```

Luego hay que asociarle un archivo de imagen, por ejemplo un bmp, png o un jpg.

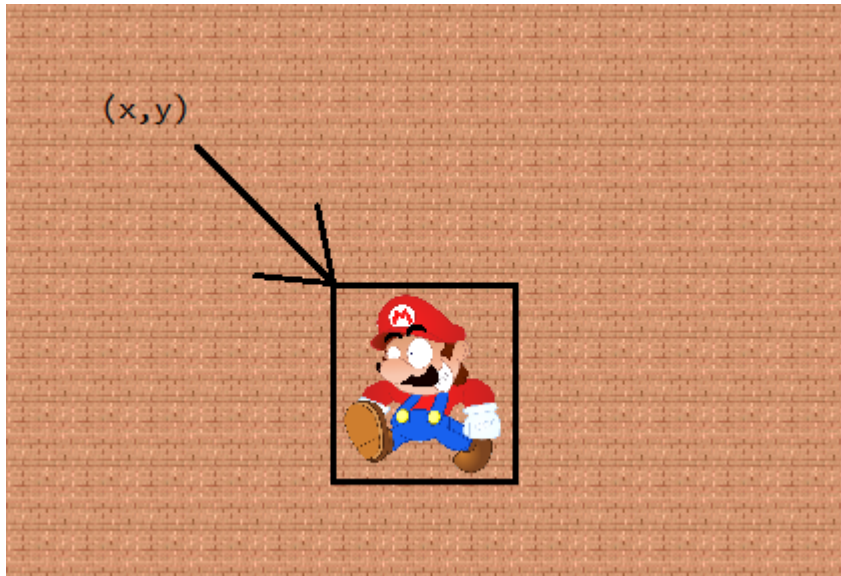
```
img_ball.src = 'ball.png';
```

Por último ya podemos dibujar en el canvas:

```
ctx.drawImage(img_ball, 0, 0);
```

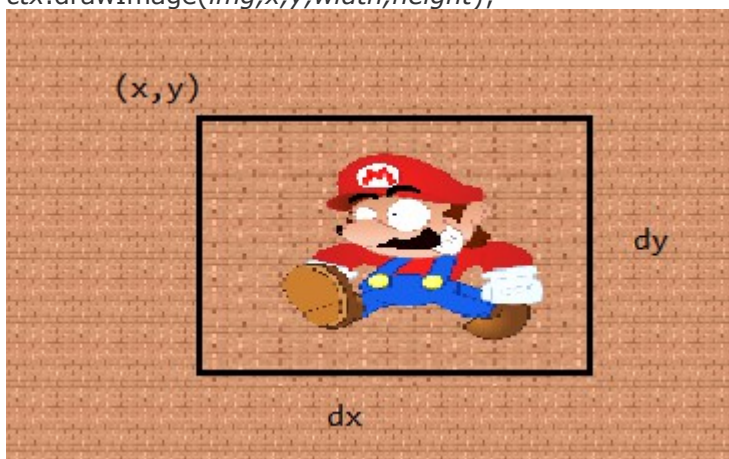
La función `drawImage` se puede llamar de diversas maneras

```
ctx.drawImage(img,x,y);
```



`(x,y)` es la posición donde queremos dibujar la imagen. El tamaño lo toma por defecto de las dimensiones de la imagen en el archivo.

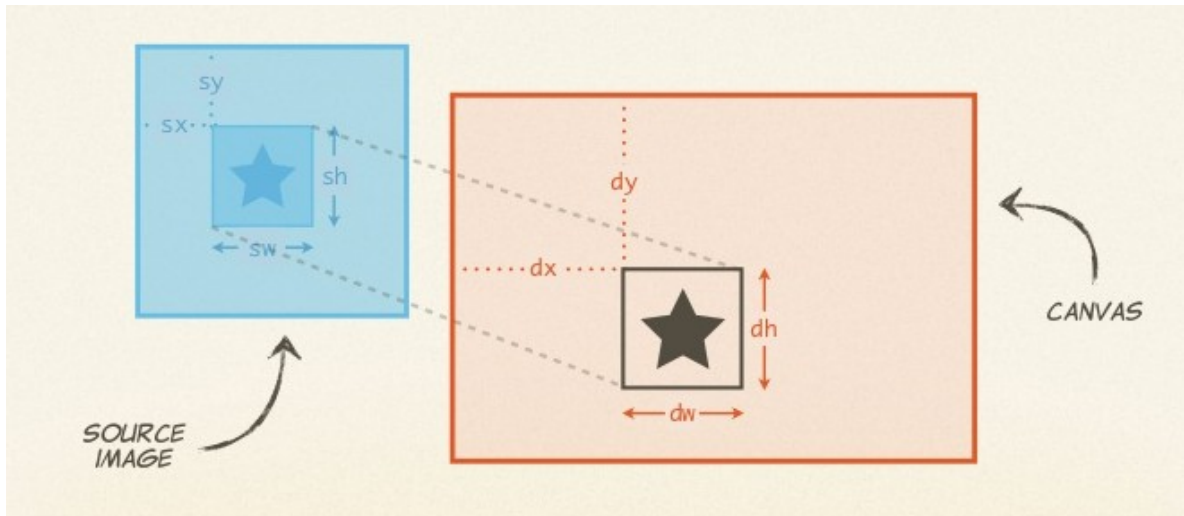
```
ctx.drawImage(img,x,y,width,height);
```



A diferencia del anterior le podemos pasar un ancho y un alto específico. En este caso la imagen será escalada (achicada o agrandada) para que se ajuste al

tamaño de salida. Si la relación entre el ancho y el alto que le pasamos a la función es diferente a la del archivo original, la imagen se podrá ver deformada.

```
ctx.drawImage(img,sx,sy,swidth,sheight,x,y,width,height);
```



Esta forma permite dibujar solo una parte de la imagen del archivo. Es sumamente útil para manejar sprites los cuales están todos almacenados en el mismo objeto imagen.

Ejemplo.



```

1. <!DOCTYPE HTML>
2. <html>
3. <head>
4. <script type="text/javascript">
5.   var canvas;
6.   var ctx;
7.   var img_ball = new Image();
8.
9.   function RenderLoop()
10.  {
11.    if(canvas.getContext)
12.    {

```

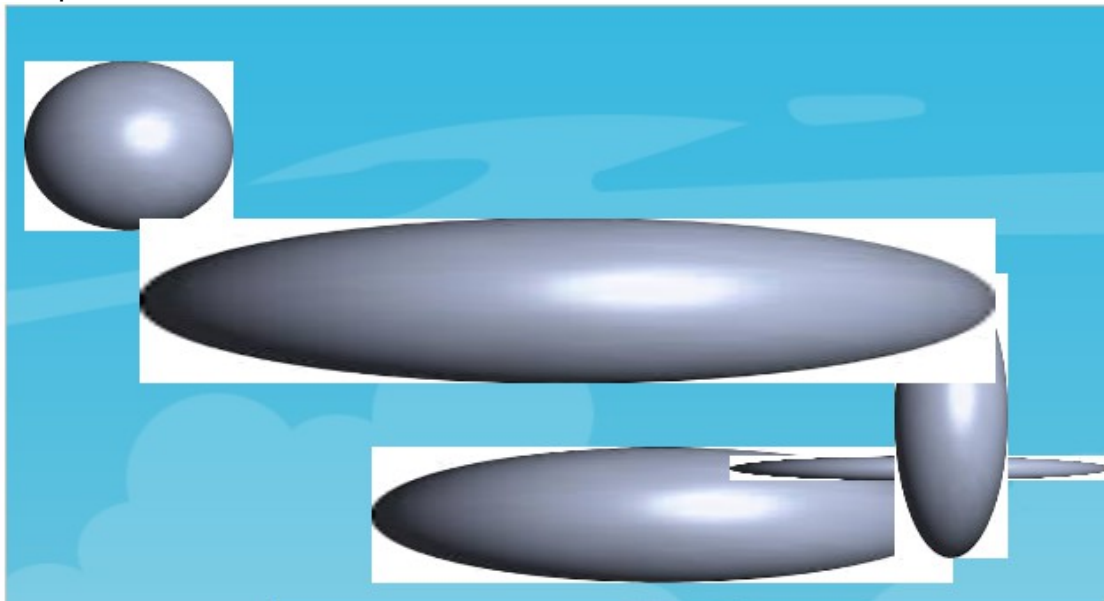
```

13.         // borro la pantalla
14.         ctx.fillStyle = 'rgba(255,255,255,255)';
15.         ctx.fillRect(0,0,2000,2000);
16.         ctx.drawImage(img_ball,100, 50, 30,30);
17.     }
18. }
19.
20. function main()
21. {
22.     canvas = document.getElementById('mycanvas');
23.     ctx = canvas.getContext('2d');
24.     img_ball.src = 'ball.png';
25.     setInterval(RenderLoop, 1000);
26. }
27. </script>
28. </head>
29. <body onload="main();">
30.     <canvas id="mycanvas" width="200" height="100" style="border:1px solid #c3c3c3;"></canvas>
31. </body>
32. </html>

```

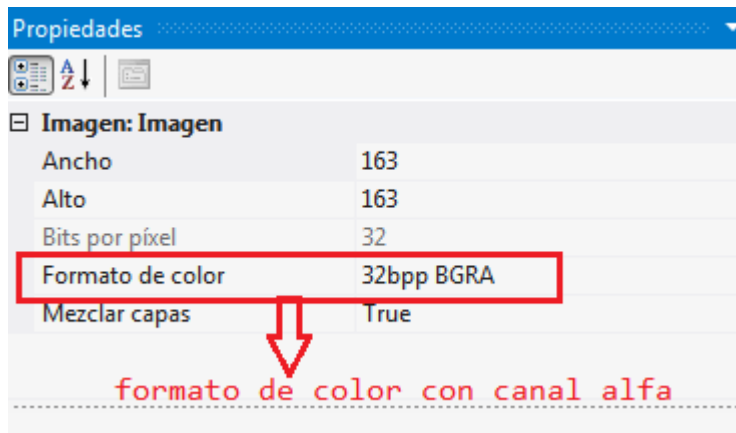
Imágenes transparentes.

Si la imagen es un bmp (bitmap) con cierto fondo y la queremos dibujar sobre otra imagen de fondo, el resultado no va a ser el esperado como se ve en la siguiente pantalla:

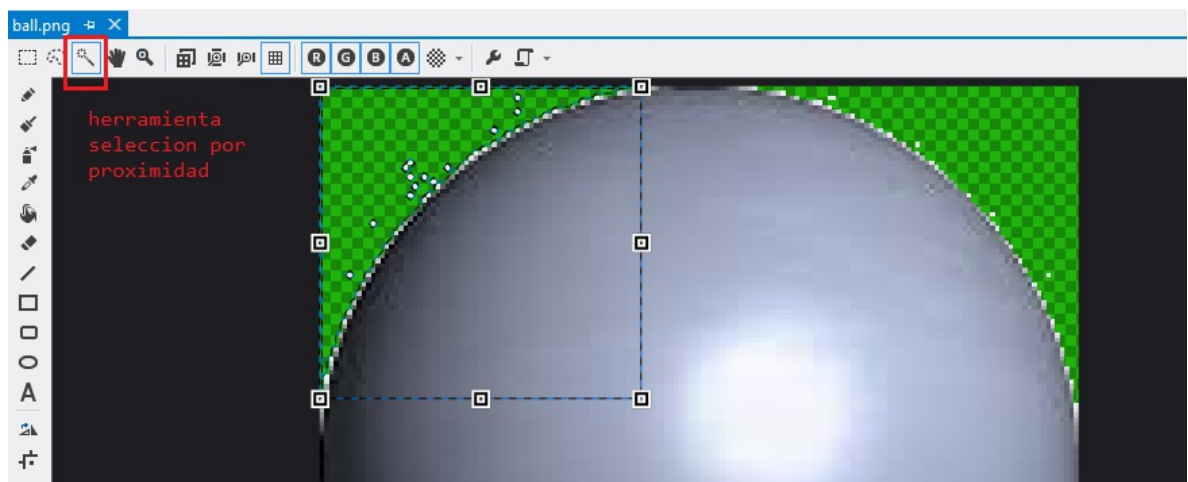


Para que la imagen se pueda dibujar bien sobre otro fondo, el formato del archivo tiene que ser rgba, donde el A indica que existe un canal de transparencia. Ese canal, no es un color como el resto de los canales (rojo, verde y azul) si no que se trata de un valor de opacidad, que puede ir desde 0 que indica que el pixel es completamente transparente y por ende queda dibujado el fondo, hasta 255 que indica que el pixel es totalmente opaco y por ende pisa el fondo. Para otros valores intermedios se van a mezclar el color de fondo con el color de la imagen, en un proceso que se llama alpha - blending.

Para editar una imagen y darle transparencia hay que usar un formato que soporte el canal alpha como el png:



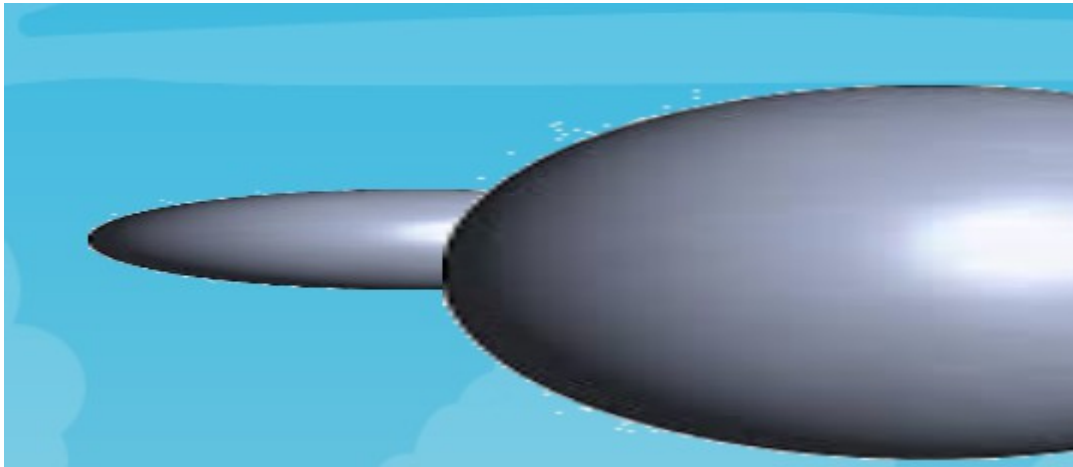
Y luego editar la imagen para definir que es lo que se va a ver opaco y que es transparente. Muchas veces ayuda la herramienta "varita mágica", que la tienen todos los editores de imágenes, y que permite seleccionar una región de la imagen en base a que es similar el color dentro de ciertos parámetros. Usando el canal alfa se puede seleccionar la parte de la imagen que no queremos que se dibuje y borrarla:



La parte de la imagen que aparece verde es el canal alfa, que indica que esa parte está transparente. Distintos editores de imágenes tienen diferentes normas para visualizar el canal alfa. Pero el concepto es siempre el mismo.

La imagen con canal alfa se puede combinar con el fondo, y esa combinación de imagen de colores + el canal alfa de transparencia, se la suele llamar "sprite".

Imagen combinada con el fondo, gracias al canal alfa:



Atlas de imágenes

Es muy común que un grupo de sprites que se refieren a la misma animación o inclusive todos los sprites de un juego, se empaqueten en una única imagen. El motivo es ahorrar tiempo de carga de la imagen, y esta relacionado con el funcionamiento interno de las placas de videos.

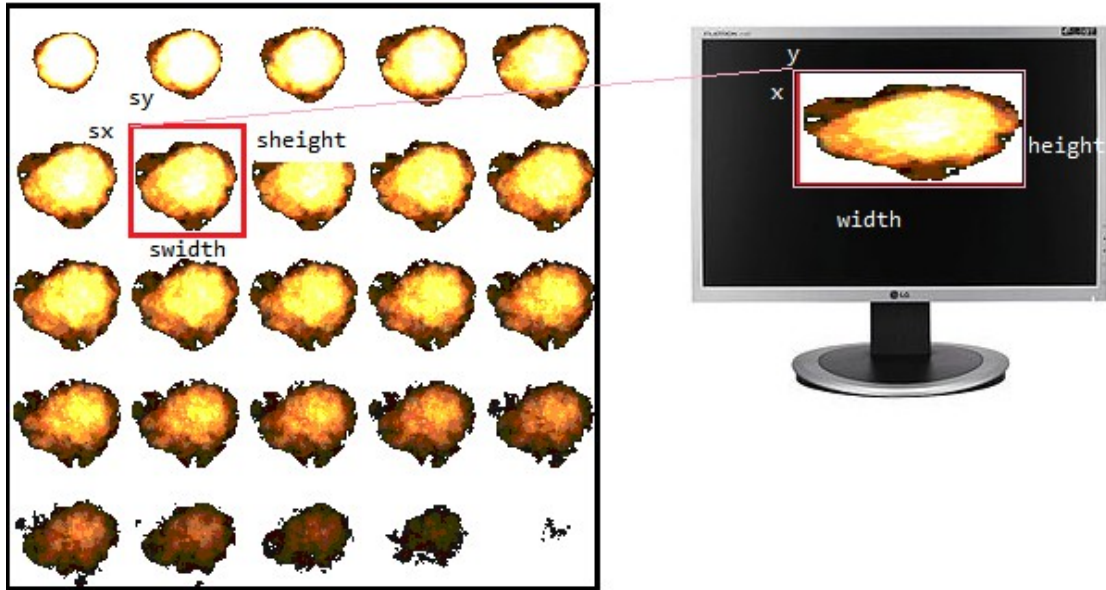
Ejemplo de sprites de una animación



Para poder utilizar esta técnica es preciso dibujar solo una parte específica de la imagen total. Y es justamente por este motivo que la función `drawImage` se puede llamar con estos parámetros:

```
ctx.drawImage(img,sx,sy,width,height,x,y,width,height);
```


Donde sx, sy es la posición de la imagen a partir de la cual vamos a dibujar, y $swidth \times sheight$ es tamaño. Luego x, y es la posición de destino donde vamos a dibujar, y $width \times height$ el tamaño.



Transformaciones.

Como la función `drawImage` permite configurar la posición y el tamaño del sprite, tenemos resuelto lo que sería la traslación y escalado. Sin embargo muchas veces tenemos que dibujar los sprites rotados en distintos ángulos. La idea es no tener que volver a dibujar el mismo sprite rotado si no poder hacerlo por código dentro del script.

Las transformaciones de rotación se estudian en los cursos de computación gráfica, ahora solo vamos a dar una forma simple de dibujar una imagen rotada en un cierto ángulo.

```

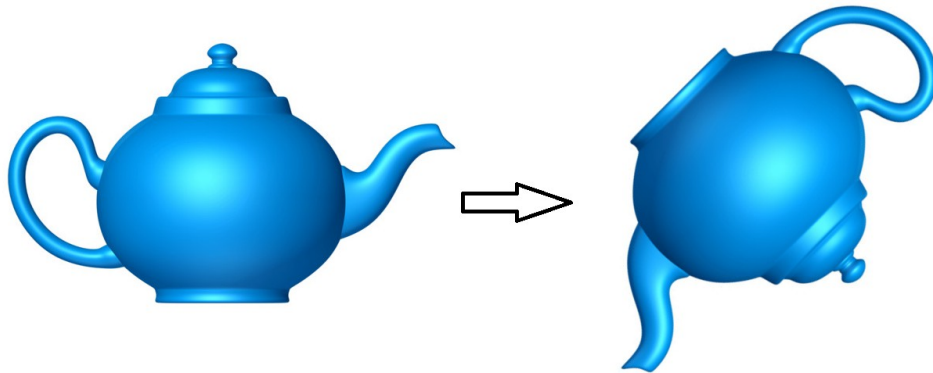
1. var x = canvas.width / 2;
2. var y = canvas.height / 2;
3. var width = image.width;
4. var height = image.height;
5.
6. ctx.save();
7. ctx.translate(x, y);
8. ctx.rotate(angleInRadians);
9. ctx.drawImage(image, -width / 2, -height / 2, width, height);
10. ctx.restore();

```

La instrucción `ctx.save()` sirve para salvar temporalmente el estado del context. Luego la instrucción `ctx.translate(x,y)` hace que el origen de coordenadas, usualmente el 0,0, sea cambiado por x,y. A partir de ahora si dibujamos algo en la posición 0,0 en lugar de salir arriba a la izquierda como siempre, va a salir en la posición x,y que indicamos.

Luego, la instrucción `ctx.rotate (an)` hace como que la pantalla gira un cierto ángulo en torno al origen (es decir el x,y). A partir de ahí todo lo que dibujemos va salir rotado.

Luego dibujamos la imagen, el `-width/2` y `-height/2` es para el centro de la imagen coincida con el origen, y eso hace que la imagen rote con respecto a su centro. Si la dibujásemos en otro lado rotaría con respecto a otro punto.

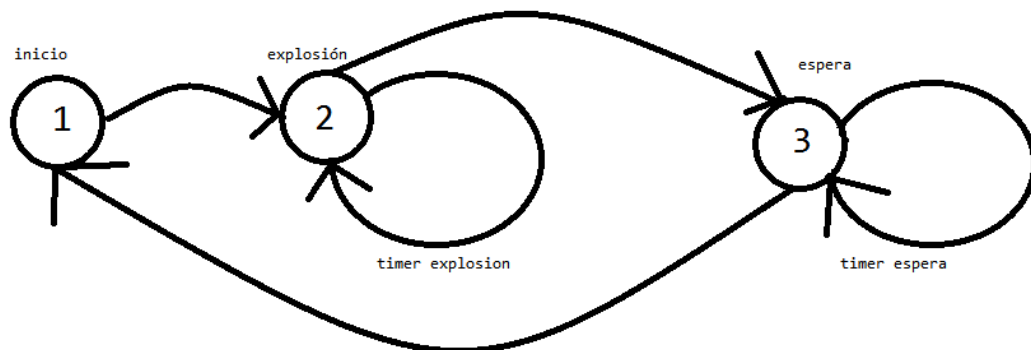


Transformación de rotación.

Timers y FSM.

En la programación de simulaciones surge naturalmente la necesidad de controlar eventos que dependen del tiempo transcurrido. Por ejemplo queremos simular una explosión que aparezca en una posición al azar y que dure 2 segundos. Al finalizar la explosión que espere un segundo y que vuelva a comentar el ciclo de una nueva explosión.

En este ejercicio es preciso controlar la secuencia de la explosión en base al tiempo transcurrido. Además también surge el concepto de estado de simulación. Muchas veces resulta útil hacer un grafo de la situación, en este caso podría ser algo así:



FSM para generar una secuencia de explosiones

Esta simulación tiene 3 estados, un estado inicial (1), un estado de explosión (2) y un estado de espera (3).

La simulación comienza en el estado 1 e inmediatamente genera una explosión en un lugar aleatorio pasando al estado 2).

Una vez que esta en el estado 2 tiene que esperar el tiempo que tarde la simulación de la explosión. Para poder controlar eso se usa una variable llamada timer, que no es otra cosa que un cronómetro que tenemos que actualizar cada vez que se llame al RenderLoop.

Cada elemento de la simulación que tenga un determinado tiempo tiene que tener asociado un timer.

Por ejemplo, para crear un explosión, podemos usar un timer así:

```
var timer_explosion = 0;
```

El timer esta inicializado en cero que indica que no hay ninguna explosión en este momento. Cuando tenemos que crear la explosión, ponemos el timer en el tiempo total de la explosión, por ejemplo:

```
1. if(estado==1)
2. {
3.     // Creo una explosión
4.     timer_explosion = 2;        // 2 segundos de explosión
5.     // y paso al estado explosión
6.     estado = 2;
7. }
```

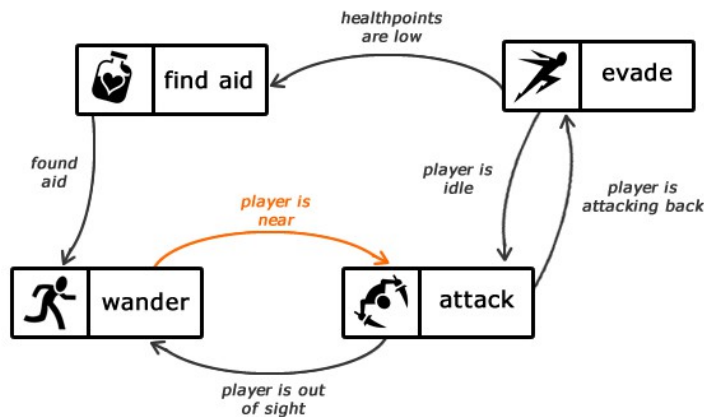
También tenemos que actualizar los distintos timers, por ejemplo:

```
1. if(timer_explosion>0)
2. {
3.     // hay una explosion en este momento
4.     dibujar_explosion(timer_explosion);
5.
6.     timer_explosion -= elapsed_time;
7.     if(timer_explosion<=0)
8.     {
9.         // termino la explosion
10.        timer_explosion = 0;        // Reseteo el timer
11.        // y paso al estado 3 de espera
12.        estado = 3;
13.        // inicio un timer de espera
14.        timer_espera = 1;        // 1 segundo de espera.
15.    }
16. }
```

Para dibujar la explosión vamos a usar el parámetro del timer, para saber en que etapa de la animación estamos, es decir si `timer_explosion == 0` es porque llegamos al final de la animación y si `timer_explosion == 2`, es porque recién empieza. Haciendo un poco de álgebra si el tiempo total de la animación es `total_explosion`, y además tenemos un total de `cant_sprites` de sprites en la animación, podemos usar esta fórmula:

```
1. nro_sprite = (total_explosion - timer_explosion) / total_explosion * cant_sprites | 0;
```

El concepto de máquina de estados es fundamental para la simulación de fenómenos físicos y para la programación de video juegos. Se llama Finite-State Machine o FSM a un modelo de computación basado en una máquina hipotética que está hecha de uno o mas estados. En un determinado momento la máquina puede estar en un único estado, de tal forma que la máquina tiene que pasar de un estado a otro para funcionar y realizar alguna tarea. Las FSM son usadas para representar y organizar el flujo del programa en los videojuegos, especialmente en las simulaciones físicas y en la inteligencia artificial.

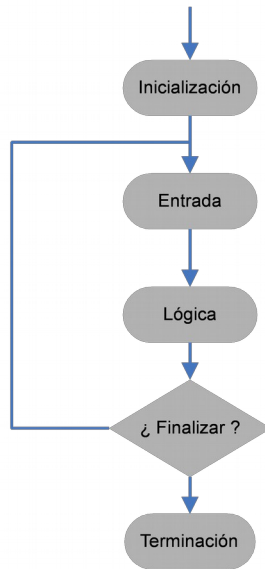


Ejemplo de FSM representando el comportamiento del enemigo.

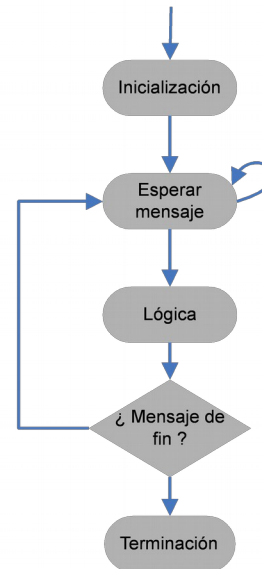
Modelo Orientado a Eventos.

La programación dirigida por eventos es un modelo de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.

Al comenzar la ejecución del programa se llevarán a cabo las inicializaciones y demás código inicial y a continuación el programa quedará bloqueado hasta que se produzca algún evento. Cuando alguno de los eventos esperados por el programa tenga lugar, el programa pasará a ejecutar el código del correspondiente administrador de evento. Por ejemplo, si el evento consiste en que el usuario ha hecho clic en el botón de disparar en un juego fps, se ejecutará el código del administrador de evento, que será el que haga que la personaje dispare.



Modelo orientado al control



Modelo orientado a eventos

Entonces en este tipo de programación, los scripts se dedican a esperar a que el usuario "haga algo" (que pulse una tecla, que mueva el ratón, que cierre la ventana del navegador). A continuación, el script responde a la acción del usuario normalmente procesando esa información y generando un resultado.

Los eventos hacen posible que los usuarios transmitan información a los programas. JavaScript define numerosos eventos que permiten una interacción completa entre el usuario y las páginas/aplicaciones web. La pulsación de una tecla constituye un evento, así como pinchar o mover el ratón, seleccionar un elemento de un formulario, redimensionar la ventana del navegador, etc.

JavaScript permite asignar una función a cada uno de los eventos. De esta forma, cuando se produce cualquier evento, JavaScript ejecuta su función asociada. Este tipo de funciones se denominan "event handlers" o "manejadores de eventos".

Tipos de Eventos

El nombre de cada evento se construye mediante el prefijo **on**, seguido del nombre en inglés de la acción asociada al evento. Así, el evento de pinchar un elemento con el ratón se denomina **onclick** y el evento asociado a la acción de mover el ratón se denomina **onmousemove**.

La siguiente tabla resume algunos de los eventos definidos por JavaScript:

Evento	Descripción
<code>onclick</code>	Pinchar y soltar el ratón
<code>ondblclick</code>	Pinchar dos veces seguidas con el ratón
<code>onkeydown</code>	Pulsar una tecla (sin soltar)
<code>onkeypress</code>	Pulsar una tecla
<code>onkeyup</code>	Soltar una tecla pulsada
<code>onload</code>	La página se ha cargado completamente
<code>onmousedown</code>	Pulsar (sin soltar) un botón del ratón
<code>onmousemove</code>	Mover el ratón
<code>onmouseup</code>	Soltar el botón que estaba pulsado en el ratón

Manejadores de eventos

Las funciones o código JavaScript que se definen para cada evento se denominan "manejador de eventos". Un evento de JavaScript por sí mismo carece de utilidad. Para que los eventos resulten útiles, se deben asociar funciones o código JavaScript a cada evento. De esta forma, cuando se produce un evento se

ejecuta el código indicado, por lo que la aplicación puede responder ante cualquier evento que se produzca durante su ejecución.

Para ello se usa la función JS del objeto documento `addEventListener`. Por ejemplo la siguiente línea indica que cuando el usuario presione una tecla hacia abajo se va a llamar a la función `doKeyDown`.

```
document.addEventListener( "keydown", doKeyDown, true);
```

Luego tenemos que escribir la función propiamente dicha, en este caso `doKeyDown`:

```
1. function doKeyDown(e)
2. {
3.     if(e.keyCode==80)
4.     {
5.         game_paused = !game_paused;
6.     }
7. }
```

Normalmente, los manejadores de eventos requieren información adicional para procesar sus tareas. Si una función por ejemplo se encarga de procesar el evento `onclick`, quizás necesite saber en que posición estaba el ratón en el momento de pinchar el botón.

No obstante, el caso más habitual en el que es necesario conocer información adicional sobre el evento es el de los eventos asociados al teclado. Normalmente, es muy importante conocer la tecla que se ha pulsado, por ejemplo para diferenciar las teclas normales de las teclas especiales (ENTER, tabulador, Alt, Ctrl., etc.).

JavaScript permite obtener información sobre el ratón y el teclado mediante un objeto especial llamado `event`. Desafortunadamente, los diferentes navegadores presentan diferencias muy notables en el tratamiento de la información sobre los eventos.

La principal diferencia reside en la forma en la que se obtiene el objeto `event`. Internet Explorer considera que este objeto forma parte del objeto `window` y el resto de navegadores lo consideran como el único argumento que tienen las funciones manejadoras de eventos.

Aunque es un comportamiento que resulta muy extraño al principio, todos los navegadores modernos excepto Internet Explorer crean mágicamente y de forma automática un argumento que se pasa a la función manejadora, por lo que no es necesario incluirlo en la llamada a la función manejadora. De esta forma, para utilizar este "argumento mágico", sólo es necesario asignarle un nombre, ya que los navegadores lo crean automáticamente.

En resumen, en los navegadores tipo Internet Explorer, el objeto `event` se obtiene directamente mediante `window.event`. Por otra parte, en el resto de navegadores, el objeto `event` se obtiene mágicamente a partir del argumento que el navegador crea automáticamente:

```
1. // en internet explorer :
2. var evento = window.event;
3.
4. // en el resto de los navegadores
5. function manejadorEventos(elEvento) {
6.     var evento = elEvento;
7. }
```

Si se quiere programar una aplicación que funcione correctamente en todos los navegadores, es necesario obtener el objeto event de forma correcta según cada navegador. El siguiente código muestra la forma correcta de obtener el objeto event en cualquier navegador:

```
1. function manejadorEventos(elEvento) {
2.     var evento = elEvento || window.event;
3. }
```

Una vez obtenido el objeto event, ya se puede acceder a toda la información relacionada con el evento, que depende del tipo de evento producido.

Información sobre los eventos de teclado

De todos los eventos disponibles en JavaScript, los eventos relacionados con el teclado son los más incompatibles entre diferentes navegadores y por tanto, los más difíciles de manejar. En primer lugar, existen muchas diferencias entre los navegadores, los teclados y los sistemas operativos de los usuarios, principalmente debido a las diferencias entre idiomas.

Además, existen tres eventos diferentes para las pulsaciones de las teclas (onkeyup, onkeypress y onkeydown). Por último, existen dos tipos de teclas: las teclas normales (como letras, números y símbolos normales) y las teclas especiales (como ENTER, Alt, Shift, etc.)

Cuando un usuario pulsa una tecla normal, se producen tres eventos seguidos y en este orden: onkeydown, onkeypress y onkeyup. El evento onkeydown se corresponde con el hecho de pulsar una tecla y no soltarla; el evento onkeypress es la propia pulsación de la tecla y el evento onkeyup hace referencia al hecho de soltar una tecla que estaba pulsada.

La mejor forma de tomar información sobre eventos de teclado es directamente sobre onkeydown y onkeyup, es decir NO utilizar onkeypress, y utilizar la propiedad event.KeyCode que tiene el código interno de la tecla.

A continuación una lista de las teclas más comunes

Key	Code	Key	Code	Key	Code
backspace	8	e	69	numpad 8	104
tab	9	f	70	numpad 9	105
enter	13	g	71	multiply	106
shift	16	h	72	add	107
ctrl	17	i	73	subtract	109
alt	18	j	74	decimal point	110
pause/break	19	k	75	divide	111
caps lock	20	l	76	f1	112
escape	27	m	77	f2	113
page up	33	n	78	f3	114
page down	34	o	79	f4	115
end	35	p	80	f5	116
home	36	q	81	f6	117
left arrow	37	r	82	f7	118
up arrow	38	s	83	f8	119
right arrow	39	t	84	f9	120
down arrow	40	u	85	f10	121
insert	45	v	86	f11	122
delete	46	w	87	f12	123
0	48	x	88	num lock	144
1	49	y	89	scroll lock	145
2	50	z	90	semi-colon	186
3	51	left window key	91	equal sign	187
4	52	right window key	92	comma	188
5	53	select key	93	dash	189
6	54	numpad 0	96	period	190
7	55	numpad 1	97	forward slash	191
8	56	numpad 2	98	grave accent	192
9	57	numpad 3	99	open bracket	219
a	65	numpad 4	100	back slash	220
b	66	numpad 5	101	close braket	221
c	67	numpad 6	102	single quote	222
d	68	numpad 7	103		

La siguiente página permite probar la secuencia de eventos que se producen al presionar una tecla y el valor de la propiedad keycode:

<http://www.asquare.net/javascript/tests/KeyCode.html>

Información sobre los eventos del mouse.

La información más relevante sobre los eventos relacionados con el mouse es la de las coordenadas de la posición del puntero. Aunque el origen de las coordenadas siempre se encuentra en la esquina superior izquierda, el punto que se toma como referencia de las coordenadas puede variar.

De esta forma, es posible obtener la posición respecto de la pantalla del ordenador, respecto de la ventana del navegador y respecto de la propia página HTML (que se utiliza cuando el usuario ha hecho scroll sobre la página). Las coordenadas más sencillas son las que se refieren a la posición del puntero

respecto de la ventana del navegador, que se obtienen mediante las propiedades clientX y clientY:

```
1. function muestraInformacion(elEvento) {  
2.     var evento = elEvento || window.event;  
3.     var coordenadaX = evento.clientX;  
4.     var coordenadaY = evento.clientY;  
5.     alert("Pos del mouse: " + coordenadaX + ", " + coordenadaY);  
6. }
```

Las coordenadas de la posición del puntero del mouse respecto de la pantalla completa del ordenador del usuario se obtienen de la misma forma, mediante las propiedades screenX y screenY:

```
1. var coordenadaX = evento.screenX;  
2. var coordenadaY = evento.screenY;
```

BIBLIOGRAFIA y ENLACES

<http://www.w3schools.com/js/default.asp>
<http://www.w3schools.com/jsref/default.asp>
<http://www.desarrolloweb.com/manuales/20/>
<http://librosweb.es/libro/javascript/>
http://www.w3schools.com/tags/ref_canvas.asp
<http://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867>