Dibujar con el elemento canvas (Parte 1)

1. INTRODUCCIÓN

El contenido **multimedia** es muy importante en el mundo web y, sin duda, las características que proporciona HTML5 referentes a este asunto han atraído la atención hacia este nuevo estándar.

Y es que uno de los objetivos del diseño de HTML5 es ser "*plugin independiente*". Es decir, establecer las bases para que sea el propio navegador el que reproduzca contenido multimedia, como animaciones, vídeo, audio, etc. sin la necesidad de utilizar programas o complementos externos.

Para ello, la especificación de HTML5 incluye varias **API** o interfaces de programación que los diseñadores utilizan y que los fabricantes de software implementan en sus navegadores.

En el caso del elemento **canvas**, se utiliza para dibujar gráficos a través de un lenguaje de *scripting* como JavaScript. No solo podremos dibujar, sino también interactuar con esos gráficos, pero siempre mediante la API de HTML5.

Esto hace que sea adecuado para el diseño de juegos, animaciones, composiciones de fotografías, etc.

El elemento **canvas** representa un lienzo o zona rectangular donde dibujar. Sin embargo, esta idea no es nueva, ya que tecnologías como **Adobe Flash** o **Microsoft Silverlight** proporcionan una funcionalidad similar.

Entonces, ¿por qué se añade **canvas** en la especificación del HTML5?

Pues porque esas tecnologías alternativas son propietarias y el estándar se basa en tecnologías **abiertas**, que pueden ser implementadas libremente por los fabricantes de los navegadores.

¿Esto quiere decir que el elemento **canvas** de HTML5 sustituirá a **Flash**?

Hay opiniones para todos los gustos, pero lo que está claro es que actualmente **Flash** es una tecnología mucho más madura que HTML5 y que seguirá utilizándose durante los próximos años.

Sin embargo, el hecho de que algunos dispositivos tan populares como el **iPhone** o el **iPad** no admitan Flash ha provocado que se avance en la adaptación del HTML5.

Por ello se hace importante aprender a trabajar con el elemento canvas y las API para vídeo y sonido de HTML5.

Empecemos pues con el elemento **canvas**.

2. UNA INTERFAZ DE DIBUJO 2D

Como se ha comentado, el elemento **canvas** representa un lienzo o zona rectangular en la página web donde dibujar. Para ello, se introduce la nueva etiqueta HTML5 **<canvas>**.

Dicha etiqueta admite los atributos **width** y **height** para establecer sus dimensiones, siendo conveniente también incluir un identificador, ya que dibujaremos en el lienzo utilizando JavaScript.

Si no se establecen los atributos **width** y **height**, entonces se crea un lienzo de **300** píxeles de anchura y **150** píxeles de altura.

```
<canvas id="lienzo1" width="300" height="300">
Contenido alternativo al elemento canvas.
</canvas>
```

Puedes incluir cualquier código HTML entre las etiquetas de inicio y fin. Ese código solo será interpretado por los navegadores que no son compatibles con canvas.

Todo esto lo podemos ver en este código. Fíjate que entre la etiqueta de apertura y de cierre de **canvas**, se incluye en este caso un determinado texto.

Ese texto es el contenido alternativo que visualizarán los navegadores que no son compatibles con este elemento de HTML5.

Una vez hemos creado el elemento en la página web, podemos manipular su superficie mediante la API de HTML5. Es decir, mediante **JavaScript**:

```
<script type="text/javascript">
function dibujar() {
   var canvas = document.getElementById("lienzo1");
   if (canvas.getContext)
   {
      var contexto = canvas.getContext("2d");
      contexto.beginPath();
      contexto.moveTo(10, 10);
      contexto.lineTo(100, 100);
      contexto.stroke();
   }
}
window.onload = dibujar;
</script>
```

Con la primera línea de la función, seleccionamos el elemento **canvas** de la página web. Fíjate que hemos empleado el método del DOM **getElementById**, de ahí la importancia de establecer el identificador en el código HTML de la página.

```
var canvas = document.getElementById("lienzo1");
```

Seguidamente se comprueba si el navegador del usuario es compatible con el elemento **canvas**. Se utiliza la estrategia ya conocida de comprobar si podemos utilizar características específicas de HTML5.

```
if (canvas.getContext)
```

Por sí solo, el elemento **canvas** no aporta las condiciones para dibujar en su superficie, sino que HTML5 utiliza el concepto de "*contexto*" para ello.

Si es compatible, entonces se crea el contexto de dibujo. En este caso, para dibujar en dos dimensiones:

```
var contexto = canvas.getContext("2d");
```

Esto no debe preocuparte demasiado, ya que simplemente quiere decir que debes obtener el contexto donde realmente dibujar. Para ello, utilizarás el método **getContext**, incluyendo el nombre del mismo, que por ahora es **2d**.

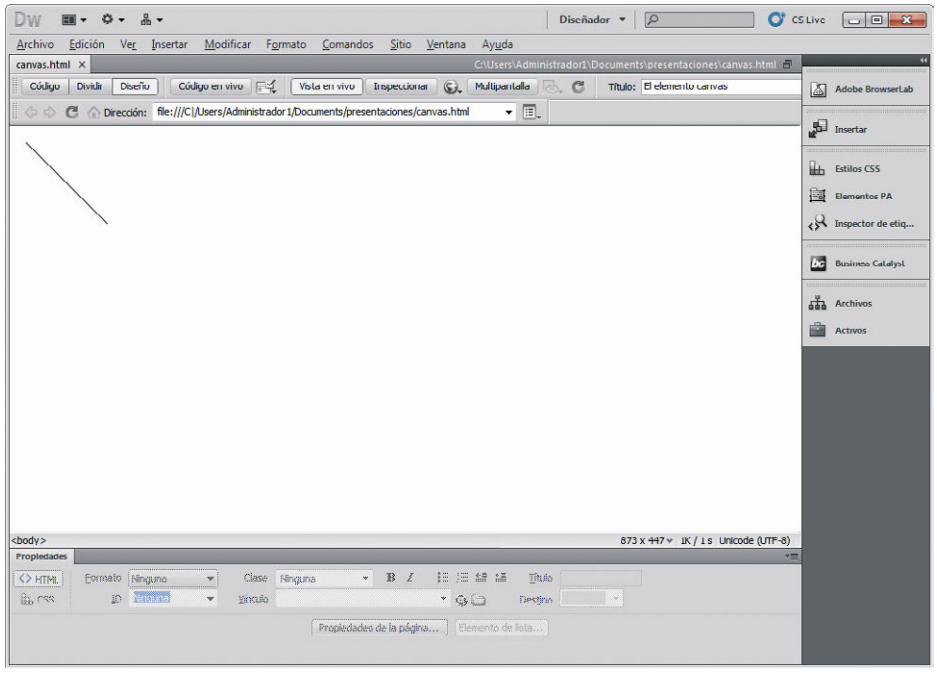
Una vez obtenido el contexto donde dibujar, podemos hacerlo con una serie de métodos. Por ejemplo, fíjate cómo se dibujaría una línea diagonal:

```
contexto.beginPath();
contexto.moveTo(10, 10);
contexto.lineTo(100, 100);
contexto.stroke();
```

No te preocupes por ahora de los métodos. Simplemente entiende que estás dibujando una línea en el contexto de dibujo **2d**.

Finalmente, asociamos este código con el evento **load** de la página, para que se ejecute en ese momento.

En la figura de la página siguiente puedes ver el resultado en la **Vista en vivo** de Dreamweaver.



¿No es muy impresionante el resultado? Pues tienes razón, el ejemplo es simple, pero intenta conseguir lo mismo en HTML 4.01 y comprobarás que no es tan sencillo. Seguramente tendrás que utilizar algo más que HTML, como Flash, CSS o JavaScript.

Este ejemplo proporciona lo básico que debes saber para trabajar con el elemento **canvas** de HTML5:

1.- Se crea un elemento en la página web a través de la etiqueta **<canvas>**.

2.- Manipulamos dicho elemento a través de **JavaScript**, para lo que lo seleccionamos con el método **getElementById** u otro similar.

3.- Obtenemos el **contexto** de dibujo, que es donde realmente podremos dibujar. Para ello, se utiliza el método **getContext**.

4.- Dibujamos utilizando los distintos métodos del contexto, como **lineTo** y **stroke**.

Fíjate que estos últimos métodos de dibujo son del contexto (en este caso el contexto "**2d**"), no del elemento **canvas** en sí.

Teniendo en cuenta estos pasos, ahora conoceremos los métodos de dibujo que podemos utilizar en el contexto **2d**.

3. DIBUJAR RECTÁNGULOS

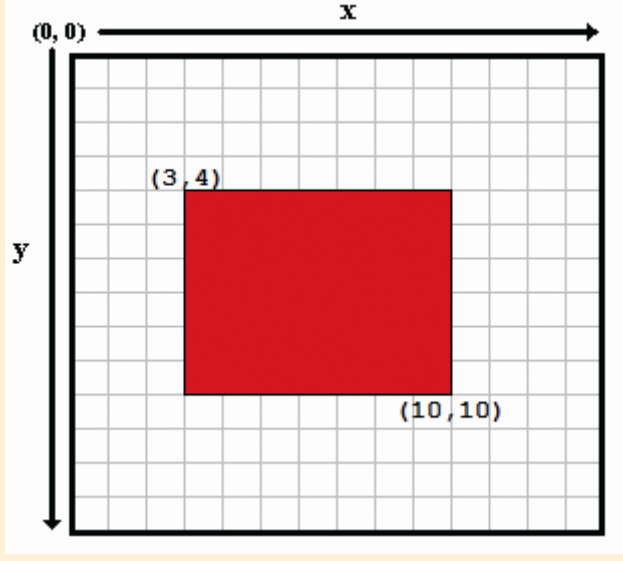
De forma similar a dibujar líneas, podemos también dibujar rectángulos. Para ello, se utilizan tres métodos del contexto **2d**:

- **fillRect**: dibuja un rectángulo relleno.
- **strokeRect**: dibuja solo el borde del rectángulo.
- **clearRect**: limpia el área correspondiente al rectángulo, haciéndola transparente.

Estos tres métodos requieren de cuatro parámetros: **x**, **y**, **anchura** y **altura**.

Con las coordenadas **x** e **y** se establece el punto inicial del rectángulo o la esquina superior izquierda; mientras que con los valores de **anchura** y **altura**, se establecen sus dimensiones.

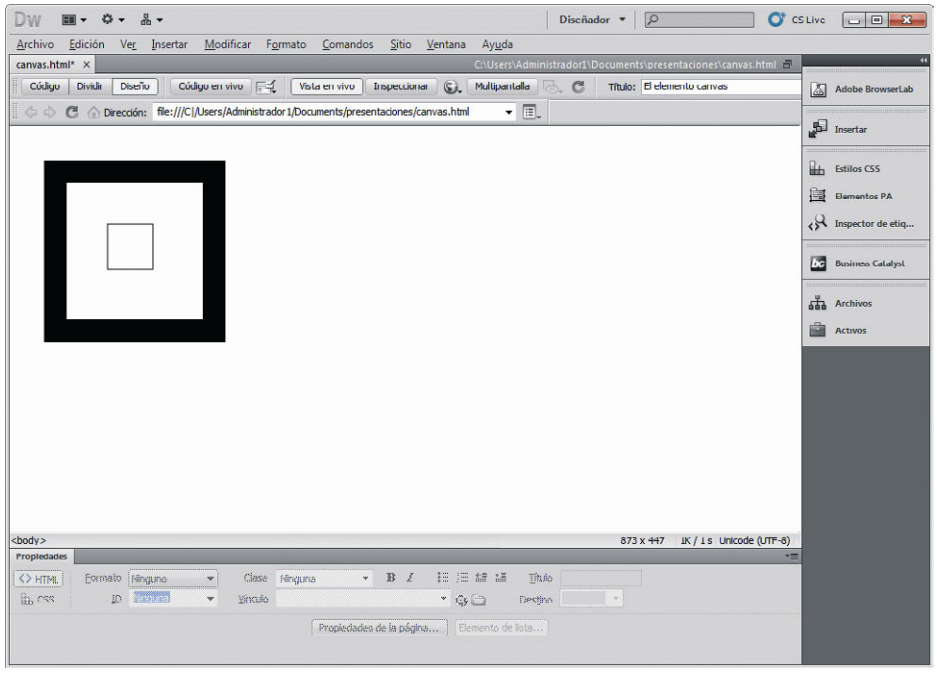
La superficie de dibujo del contexto **2d** se entiende como aquella en la que el origen de coordenadas o punto (0, 0) coincide con el borde superior izquierdo del elemento canvas, como se muestra en esta figura.



Estudia las siguientes líneas de código:

```
contexto.fillRect(30, 30, 200, 200);
contexto.clearRect(55, 55, 150, 150);
contexto.strokeRect(100, 100, 50, 50);
```

Primero dibujamos un rectángulo con el color de relleno predeterminado, que es el negro; después se limpia un área concreta del elemento **canvas**, por lo que el fondo será transparente; y, finalmente, se dibuja el borde de otro rectángulo. Aquí tienes el resultado:



En este caso realmente se han dibujado cuadrados, ya que hemos establecido el mismo valor para la anchura y altura de los mismos.

Es importante fijarse en un asunto que va a ser distinto para el resto de métodos de dibujo: los rectángulos se han dibujado inmediatamente en la superficie del **canvas**.

Es decir, que los métodos **fillRect**, **clearRect** y **strokeRect** son métodos que dibujan en el canvas sin esperar a ninguna orden más.

4. DIBUJAR TRAZADOS

A diferencia de lo que ocurre cuando utilizamos los métodos para dibujar rectángulos, el resto de métodos de dibujo del contexto **2d** (por ejemplo, para dibujar líneas, curvas o arcos) se realizan como parte de un **trazado** o *path* en inglés.

Por ello no podemos utilizar directamente los métodos de dibujo, sino que hay que realizar una serie de pasos extra:

1º- Utilizar el método **beginPath** para iniciar el trazado.

2º- Utilizar los métodos de dibujo, como **lineTo** para dibujar líneas o **arc** para dibujar arcos.

3º- Utilizar opcionalmente el método **closePath** para cerrar el trazado.

Este método cierra el trazado mediante una línea que une el último punto dibujado con el inicial, si es necesario.

4º- Finalmente, utilizar el método **stroke** o **fill** para dibujar realmente en la superficie del **canvas**. El método **stroke** solo incluye el borde de las figuras, mientras que **fill** las dibuja con relleno.

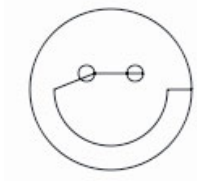
Fíjate en el siguiente código:

```
contexto.beginPath();
//círculo exterior
contexto.arc(75, 75, 50, 0, Math.PI*2, true);
//boca
contexto.arc(75, 75, 35, 0, Math.PI, false);
//ojo izquierdo
contexto.arc(60, 65, 5, 0, Math.PI*2, true);
//ojo derecho
contexto.arc(90, 65, 5, 0, Math.PI*2, true);
contexto.stroke();
```

Se trata de un trazado en el que se ha utilizado el método **arc** para dibujar arcos o círculos completos. El propósito es dibujar una cara sonriente como esta.

Como puedes ver se ha seguido los pasos anteriormente mencionados.

Sin embargo, si ejecutas el código anterior, vas a obtener este otro resultado. Vemos que se han dibujado los distintos círculos y arcos para la cara sonriente, pero aparecen una serie de líneas que los unen.



Esto ocurre porque el dibujo se realiza como si tuviéramos un lápiz presionando todo el rato el papel.

Podemos evitar esto con el método **moveTo**. Este método produce el efecto de "*levantar*" el lápiz para moverlo al lugar donde volvemos a dibujar.

```
contexto.beginPath();
//círculo exterior
contexto.arc(75, 75, 50, 0, Math.PI*2, true);
contexto.moveTo(110, 75);
//boca
contexto.arc(75, 75, 35, 0, Math.PI, false);
contexto.moveTo(65, 65);
//ojo izquierdo
contexto.arc(60, 65, 5, 0, Math.PI*2, true);
contexto.moveTo(95, 65);
//ojo derecho
contexto.arc(90, 65, 5, 0, Math.PI*2, true);
contexto.stroke();
```

Añadiendo las líneas que aparecen en negrita sí que conseguimos el trazado sin las líneas que unían los distintos círculos y el arco de la cara.

Es importante recordar que todos los métodos de dibujo, excepto los correspondientes a dibujar rectángulos, deben incluirse en un trazado. Es decir, que debe utilizarse primero el método **beginPath** y, finalmente, los métodos **stroke** o **fill**.

Estos son los métodos de dibujo que puedes utilizar en un trazado:

- **lineTo(x, y)**: dibuja una línea. Los parámetros **x** e **y** representan las coordenadas del punto final de la línea, siendo el punto inicial la posición actual.

- **arc(x, y, radio, ánguloInicial, ánguloFinal, sentido)**: para dibujar círculos o arcos.

Este método emplea cinco parámetros:

- **x** e **y** representan las coordenadas del centro del círculo,

- **radio** es la longitud (en píxeles) del radio del círculo,

- **ánguloInicial** y **ánguloFinal** definen el punto inicial y final del círculo. Su valor se establece en radianes,

- **sentido**: indica si se utiliza el sentido de las agujas del reloj (valor **false**) o el contrario (valor **true**).

Para convertir grados en radianes, podemos utilizar esta expresión:

radianes = (Math.PI / 180) * grados

Además, hay alguna expresión que se utiliza frecuentemente, como **Math.PI * 2** que representa 360 grados o **Math.PI** que son 180 grados.

- **quadraticCurveTo(cx1, cy1, x, y)**: para dibujar curvas Bézier con un único punto de control.

- **bezierCurveTo(cx1, cy1, cx2, cy2, x, y)**: para dibujar curvas Bézier con dos puntos de control.

- **rect(x, y, anchura, altura)**: para dibujar rectángulos que forman parte de un trazado. En este caso actúa como el resto de los métodos anteriores, es decir, que no se dibujará el rectángulo hasta que no utilicemos el método **stroke** o **fill** correspondiente.

La ejecución de este método implica la llamada implícita al método **`moveTo(0, 0)`**, es decir, que el punto inicial del rectángulo es el origen de las coordenadas del elemento **`canvas`**.

Recuerda que son métodos del contexto **2d**. Cuando se añadan otros contextos en la especificación del elemento **canvas**, existirán otros métodos que podremos utilizar.

5. COLORES DE TRAZO Y DE RELLENO

Hasta ahora, todos los ejemplos que hemos visto utilizan el trazo y relleno predeterminados.

Hemos visto que esto se refleja en el grosor y estilo de las líneas o bordes de las figuras, así como en el uso del color negro tanto para el contorno como para el relleno.

Bien, existen algunos métodos y propiedades del contexto **2d** que permiten aplicar otros estilos tanto al trazo de las figuras dibujadas como a su relleno.

Por ejemplo, para establecer un determinado **color** podemos utilizar las propiedades **strokeStyle** para el color del trazo y **fillStyle** para el color del relleno.

El valor esperado para ambas propiedades es un color, que se utilizará a partir del momento en que se establezca y para todas las operaciones de dibujo.

Por lo tanto, deberemos conocer cómo establecer ese valor de color y es sencillo porque se realiza de la misma forma que en las hojas de estilo en cascada **CSS**:

```
contexto.strokeStyle = "#ff0000";
```

De esta forma estamos estableciendo el color **rojo** para el contorno de las figuras.

Y con:

```
contexto.fillStyle = "#0000ff";
```

establecemos el color **azul** para que se utilice como color de relleno de las figuras.

Fíjate que ambas órdenes se aplicarán en las operaciones pendientes de dibujo y hasta que se indique lo contrario, independientemente de dónde las incluyamos.

Claro está que el color establecido a través de la propiedad **strokeStyle** tendrá efecto para las operaciones **stroke** y el color establecido con la propiedad **fillStyle** para las operaciones **fill**. Cuando se ejecuta una orden **stroke** o **fill** se utilizan los colores activos en ese momento.

Por eso, incluyendo estas dos órdenes en el código anterior, conseguiríamos este otro resultado.

No parece que sea el resultado buscado, ya que la idea es que el relleno solo afecte a los círculos de los ojos de la cara.

Sin embargo, como todas las órdenes de dibujo pertenecen a un mismo trazado, se aplica primero el color de trazado (strokeStyle) y después el de relleno (fillStyle).

Es fácil arreglar esto abriendo un nuevo trazado para la operación de relleno, es decir, utilizando otro método **beginPath**:

```
<script type="text/javascript">
function dibujar() {
     var canvas = document.getElementById("lienzo1");
     if (canvas.getContext)
     {
           var contexto = canvas.getContext("2d");
           contexto.beginPath();
           //círculo exterior
           contexto.arc(75, 75, 50, 0, Math.PI*2, true);
           contexto.moveTo(110,75);
           contexto.strokeStyle = "#ff0000";
           //boca
           contexto.arc(75, 75, 35, 0, Math.PI, false);
           contexto.stroke();
           contexto.beginPath();
           contexto.moveTo(65,65);
           contexto.fillStyle = "#0000ff";
```

```
            //ojo izquierdo
            contexto.arc(60, 65, 5, 0, Math.PI*2, true);
            contexto.moveTo(95,65);
            //ojo derecho
            contexto.arc(90, 65, 5, 0, Math.PI*2, true);
            contexto.fill();
     }
}
window.onload = dibujar;
</script>
```

Con este cambia sí que conseguimos el resultado deseado: dibujando primero el círculo de la cara y el arco de la boca; y rellenando los círculos correspondientes a los ojos.

Vemos que para dibujar con el elemento **canvas** se hace necesario escribir el código para representar las distintas figuras geométricas a través de todos estos métodos y propiedades.

Esto representa un esfuerzo importante de desarrollo a diferencia de lo que ocurre con otras tecnologías alternativas como **Flash** o **Silverlight** que disponen de completos entornos de desarrollo visual.

Sin embargo, a medida que transcurra el tiempo, es de esperar que aparezcan aplicaciones especializadas para que el dibujo se realice de forma visual y que generen automáticamente todo este código HTML5.