2.4.4.- Ejemplo para cambiar el color de relleno

Veamos ahora un ejercicio que nos sirve de ejemplo para mostrar cómo cambiar el color de relleno. En concreto vamos a rellenar el canvas con muchos cuadraditos de colores aleatorios y además haremos que cada pocos instantes se vuelva a dibujar el canvas con nuevos cuadraditos aleatorios, con lo que generaremos nuestra primera y pequeña animación.

Veamos antes que nada un par de funciones para conseguir un color aleatorio en Javascript. La primera nos ofrece un número aleatorio y la segunda, que se apoya en la primera, nos sirve para generar una cadena que especifica un color.

```
function aleatorio(inferior,superior){
numPosibilidades = superior - inferior
aleat = Math.random() * numPosibilidades
aleat = Math.floor(aleat)
return parseInt(inferior) + aleat
}
function colorAleatorio(){
return "rgb(" + aleatorio(0,255) + "," + aleatorio(0,255) + "," + aleatorio(0,255) + ")";
}
```

Ahora vamos a mostrar otra función para dibujar el lienzo de un canvas, rellenando de cuadraditos con colores aleatorios:

```
function cuadradosAleatorios()
{
for(i=0; i<300; i+=10)
{
        for(j=0; j<250; j+=10)
        {
        contexto.fillStyle = colorAleatorio();
        contexto.fillRect(i,j,10,10)
        }
}
}
```

Como se puede ver, tenemos un bucle anidado, que realiza la tarea. En cada iteración se obtiene un color aleatorio y luego se pinta un rectángulo con ese color. La función utiliza una variable global llamada "contexto", que es el contexto del canvas sobre el que estamos dibujando.

Ahora para acabar vamos a ver la función que se encargará de inicializar el contexto del canvas y definir la ejecución periódica de la función cuadrados Aleatorios() para generar la animación.

```
//variable global contexto sin inicializar
var contexto;
window.onload = function(){
//Recibimos el elemento canvas
contexto = cargaContextoCanvas('micanvas');
if(contexto){
//Si tengo el contexto, defino la función periódica
setInterval("cuadradosAleatorios(contexto)", 200);
}
}
```

Todo esto junto hace que consigamos una animación en el canvas, pues se invoca a la función cuadradosAleatorios() cada 200 milisegundos, lo que genera dibujos aleatorios distintos cada poco tiempo.

2.4.5.- Ejemplo para cambiar el color del trazado

Hemos hecho el ejemplo anterior con una ligera modificación. En este segundo caso, en lugar de rellenar los rectángulos de color, vamos a dibujar sólo el trazado. Por supuesto, en cada paso del bucle se cambiará el color de trazado, en lugar del color de relleno. Además, los rectángulos cuyo trazado estamos dibujando serán un poco menor.

La única función que tiene cambios con respecto al ejemplo anterior es cuadradosAleatorios():

```
function cuadradosAleatorios(){
for(i=0; i<300; i+=10){
for(j=0; j<250; j+=10){
contexto.strokeStyle = colorAleatorio();
contexto.strokeRect(i,j,5,5)
}
}
}
```

Ver ejemplo

Parte 3:

Dibujar caminos en Canvas

A través de los caminos se pueden dibujar todo tipo de figuras en el lienzo de Canvas.

Vemos todos los tipos de caminos que existen y diferentes ejemplos prácticos.

3.1.- Caminos en Canvas del HTML 5

Veamos cómo realizar dibujos en un canvas con las funciones para caminos, que permiten la creación de estructuras más complejas.

El Canvas es uno de los elementos más novedosos del HTML 5, que ya comenzamos a explicar en el artículo de Introducción a Canvas. En anteriores artículos vimos ejemplos sobre diversos dibujos en un elemento canvas, como los rectángulos. Ahora vamos a continuar aprendiendo cómo dibujar estructuras diversas por medio de los caminos.

En canvas existen diversas funciones que nos pueden servir para dibujar siluetas a nuestro antojo, que se tienen que utilizar de manera complementaria. El proceso pasa por situarse en un punto del lienzo, luego definir cada uno de los puntos por los que pasa nuestro camino y luego pintar de color dentro, o simplemente dibujar la línea que pasaría por todos esos puntos. En este artículo veremos cómo rellenar de color todo el área que queda definida por un camino.

Veamos para empezar un resumen de algunas de las funciones disponibles para hacer caminos, las que que utilizaremos durante el presente artículo.

3.1.1.- Función beginPath()

Esta función sirve para decirle al contexto del canvas que vamos a empezar a dibujar un camino. No tiene ningún parámetro y por si sola no hace ninguna acción visible en el canvas. Una vez invocada la función podremos empezar a dibujar el camino añadiendo segmentos para completarlo con las diferentes funciones de caminos.

Nota: Las funciones beginPath() y siguientes en realidad son métodos del objeto de contexto del canvas. Este objeto que mantiene el contexto del canvas lo tenemos que extraer nosotros por medio de Javascript, a partir del elemento canvas donde deseemos dibujar. Cómo trabajar y extraer el contexto de un canvas fue ya explicado en el artículo Ejemplo de dibujo con el API de canvas.

Según las pruebas realizadas, podríamos iniciar un camino sin utilizar antes beginPath(), puesto que el efecto a primera vista es el mismo que si no lo invocamos (entiendo que el navegador lo invoca por nosotros al empezar a utilizar funciones de caminos en canvas). No obstante, debe ser recomendable hacer las cosas correctamente e invocarlo antes de comenzar un camino.

3.1.2.- Función moveTo()

Sirve para mover el puntero imaginario donde comenzaremos a hacer el camino. Esta función no dibuja nada en si, pero nos permite definir el primer punto de un camino. Llamar esta función es como si levantásemos el lápiz del lienzo y lo trasladásemos, sin pintar, a otra posición.

Recibe como parámetro los puntos x e y donde ha de moverse el puntero para dibujo. Para saber cuál es el punto donde deseamos movernos (x,y), Recordar que el eje de coordenadas del canvas es la esquina superior izquierda.

3.1.3.- Función lineTo()

Esta función provoca que se dibuje una línea recta, desde la posición actual del puntero de dibujo, hasta el punto (x,y) que se indique como parámetro. El método lineTo(), por tanto es como si posáramos el lápiz sobre el lienzo en la posición actual y arrastrásemos, dibujando una línea recta, hasta el punto donde se definió al invocar el método.

La posición actual del camino la podemos haber indicado previamente con un moveTo(), o donde hayamos terminado una línea dibujada anteriormente. Si no se indicó antes una posición de nuestro puntero de dibujo, lineTo() no dibuja ninguna línea, sino que se tendrá en cuenta las coordenadas enviadas como parámetro para posicionar tan solo el puntero de dibujo allí. Dicho de otra manera, si no se dijo dónde empezar el dibujo, o no se ha dibujado ningún otro segmento en el camino anteriormente, lineTo() será equivalente a moveTo().

3.1.4.- Función fill()

Este método del contexto del canvas sirve para rellenar de color el área circunscrita por un camino. Para rellenar de color un camino, el camino tendría que estar cerrado, por lo que, si no lo está, automáticamente se cerrará con una línea recta hasta el primer punto del camino, es decir, donde comenzamos a dibujar. Sin embargo, si durante los distintos segmentos del camino nos dejamos algún segmento abierto, no se pintará nada.

Como decimos, si no llegamos a cerrar el camino, el método fill() lo cerrará por nosotros, pero podríamos utilizar explícitamente el método closePath() para hacerlo nosotros (closePath() lo explicaremos en futuros artículos).

3.1.5.- Ejemplo de camino sencillo

Con las funciones vistas hasta el momento ya podemos hacer unas primeras pruebas de caminos en canvas. Ahora vamos ver como podríamos realizar un rombo en el canvas, relleno de color.

```
ctx.beginPath();
ctx.moveTo(50,5);
ctx.lineTo(75,65);
ctx.lineTo(50,125);
ctx.lineTo(25,65);
ctx.fill();
```

Como se puede ver, iniciamos un camino con beginPath(). Luego hacemos un moveTo() para indicar el punto donde comenzar el camino. Posteriormente dibujamos varias líneas a diversos puntos del canvas, para acabar invocando al método fill(), con lo que rellenaremos de color el camino.

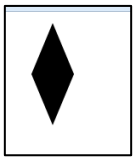
Fijarse que el camino no se había llegado a cerrar. Por lo que fill() lo cerrará por nosotros con una línea al primer punto donde comenzamos el dibujo.

Nota: Para ejecutar estas líneas de código necesitaremos una instancia del objeto contexto del canvas, para invocar todos los métodos sobre él. El objeto del canvas lo tenemos en la variable "ctx" en el código del ejemplo. En el código completo del ejercicio podremos ver la función que se podría utilizar para obtener el contexto.

3.1.6.- Código completo del ejemplo de camino

A continuación podemos encontrar el código completo de este ejemplo de construcción de un camino con el elemento Canvas del HTML 5.

```
<html>
<head>
<title>Canvas Caminos</title>
<script>
//La ya conocida función para cargar el contexto de un canvas
function cargaContextoCanvas(idCanvas){
var elemento = document.getElementById(idCanvas);
if(elemento && elemento.getContext){
var contexto = elemento.getContext('2d');
if(contexto){
return contexto;
}
}
return FALSE;
}
window.onload = function(){
//Recibimos el elemento canvas
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
ctx.beginPath();
ctx.moveTo(50,5);
ctx.lineTo(75,65);
ctx.lineTo(50,125);
ctx.lineTo(25,65);
ctx.fill();
}
}
</script>
</head>
<body>
<canvas id="micanvas" width="150" height="150">
Accede a este script con un navegador que acepte canvas del HTML 5
</canvas>
</body>
</html>
```



3.2.- Ejemplo 2 de dibujo de caminos en canvas

Segundo ejemplo sobre los caminos en el elemento canvas, donde veremos las funciones closePath() y stroke().

El elemento canvas es un lienzo donde podemos dibujar directamente con funciones Javascript, que tiene ya aplicaciones infinitas en el desarrollo de webs. Estamos explicando acerca de este elemento en el Manual de Canvas del HTML 5 y en el presente artículo vamos a seguir hablando del dibujo caminos, viendo dos nuevas funciones del API de Canvas. Conviene no obstante señalar que el tema sobre caminos en canvas lo empezamos a explicar en Caminos en Canvas del HTML 5.

En el presente artículo veremos dos nuevas funciones útiles en la creación de caminos, que son closePath(), para cerrar un camino y stroke(), para dibujar el camino realizado mediante una línea. Las dos funciones, como cualquier otra función de dibujo en el lienzo de canvas, son métodos del objeto contexto del canvas, que se debe obtener a partir del elemento canvas con las correspondientes funciones de Javascript, tal como vimos anteriormente en este manual. Veremos estas nuevas funciones para dibujo de caminos con un ejemplo, pero antes podemos explicarlas detalladamente.

3.2.1.- Función closePath()

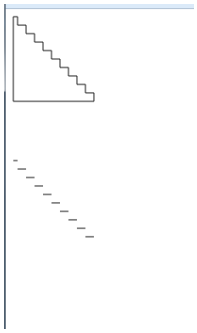
Sirve para cerrar un camino, volviendo a su punto inicial de dibujo. Recordemos que el camino tiene un punto inicial en el que nos situamos para comenzar el dibujo, con moveTo(). Luego vamos dibujando segmentos en el camino por medio de líneas que nos llevan a otros puntos del lienzo. Pues closePath() sería como dibujar una línea recta desde el punto donde se haya quedado el camino al punto inicial donde empezamos a construirlo. El método closePath() no recibe ningún parámetro.

3.2.2.- Función stroke()

Con el método stroke() podemos dibujar una línea por todo el recorrido del camino que hayamos creado por medio de sus distintos segmentos. Es similar al método fill(), explicado en el artículo anterior, con la diferencia que fill() rellenaba de color y stroke() tan solo dibuja la silueta. Además, en el caso de fill() se necesitaba tener el camino cerrado, por lo que se cerraba automáticamente si no lo habíamos hecho y stroke() realmente puede estar discontinuada, puesto que sólo es una línea lo que se dibuja y no un área.

3.2.3.- Ejemplo de camino con closePath() y stroke()

A continuación vamos a realizar otro ejemplo de dibujo con el API de canvas y utilizando funciones para la realización de caminos. Por complicarlo un poco, vamos a realizar el camino con un bucle, en el que en cada iteración dibujaremos un segmento del camino. El resultado que vamos a obtener es una especie de perfil de una escalera.



Nota: Primero cabe advertir de nuevo que para ejecutar ese código necesitamos una variable que hemos llamado "ctx" que contiene el contexto del canvas, que es sobre el que invocaremos los distintos métodos para dibujar en el canvas.

En el script comenzamos el camino con beginPath(), luego con moveTo(1,1) nos situamos en el punto donde deseamos comenzar el dibujo. A continuación realizamos un bucle for para dibujar diversas líneas en diversas coordenadas.

Acabamos haciendo una última línea con lineTo() y después un closePath() para que se dibuje una línea final hasta el punto de inicio del camino, que cerrará la silueta realizada. Con stroke() hacemos que se dibuje una línea pasando por todos los segmentos que completan el camino dibujado.

3.2.4.- Ejemplo de línea discontinua

Si habemos observado el ejemplo anterior en marcha habremos visto que en realidad hay dos ejemplos de canvas. El segundo es igual que el primero, o casi igual, con la única diferencia que el camino no está cerrado y está formado por una línea discontinua. Esto se puede hacer perfectamente con stroke(), pues para pintar líneas no es necesario que cierren completamente el camino.

3.2.5.- Ejemplo completo de dibujo de líneas con caminos en canvas

(Hacerlo)

3.3.- Caminos en canvas: ejemplo 3

Cómo dibujar un camino en canvas de HTML 5 con diversas variantes, cerrado y sin cerrar, y con o sin relleno de color.

Quizás nos estemos poniendo un poco repetitivos con los caminos en Canvas, pero es que el tema es suficientemente importante como para publicar varios artículos. Como los propios lectores de .com dicen, nunca está de más poner varios ejemplos que ayuden un poco más a asimilar los conocimientos brindados en los manuales.

En artículos anteriores ya explicamos cómo hacer caminos en canvas del HTML 5 esta ocasión vamos a dedicarnos a dibujar el mismo camino, que es un simple hexágono, pero con distintas variantes, para que las personas puedan ver las diferencias entre cerrar o no los caminos, así como rellenarlos de color.

En el presente artículo veremos las siguientes variantes de un camino con la forma de hexágono regular:

1. Camino relleno de color y con el cierre de camino no explícito.

2. Camino relleno de otro color y con el cierre de camino explícito por medio de closePath().

3. Camino sin relleno de color, sólo la línea, y sin cierre de camino

4. Camino sin relleno de color, sólo la línea de la silueta y con cierre de camino explícito.

Realmente es un mismo ejercicio con varias variantes que esperamos pueda darnos alguna pista adicional sobre el dibujo en el elemento canvas del HTML 5. Podemos ver una imagen con los cuatro ejemplos de caminos que haremos a continuación:

3.3.1.- 1.- Camino relleno sin cierre explícito

Este primer ejemplo de dibujo en un canvas creará un camino con forma de rectángulo que tendrá un relleno de color. En este caso el el camino no está cerrado, pero veremos que da un poco igual en este caso.

Para rellenar de color un camino utilizamos la el método fill() del contexto del canvas, que antes de rellenar de color hace un cierre automático del camino. De esta manera, aunque no se haya completado el camino hasta cerrarlo, al invocar ctx.fill() esta función lo cerrará por nosotros.

```
//Hexagono relleno de color, cierre de camino automático con fill
var ctx = cargaContextoCanvas('canvas1');
if(ctx){
ctx.beginPath();
ctx.moveTo(50,15);
ctx.lineTo(112,15);
ctx.lineTo(143,69);
ctx.lineTo(112,123);
ctx.lineTo(50,123);
ctx.lineTo(19,69);
ctx.fill();
}
```

3.3.2.- 2.- Camino relleno con cierre explícito

En esta segunda variante del camino del hexágono tenemos un camino que sí hemos cerrado explícitamente con el método closePath(). Sin embargo, como fill() ya se encargaba de cerrar el camino por nosotros automáticamente, no existe diferencia entre ese camino y el anterior. Es decir, para el caso de caminos con

color de relleno, es indiferente si el camino está o no cerrado, pues se cerrará automáticamente para poder rellenarse de color.

No obstante, para adornar un poco más el ejemplo, hemos optado por cambiar el color de relleno del hexágono, por medio de la propiedad fillStyle del objeto contexto del canvas.

```
//Hexagono rellenado, cierre de camino explícito con closePath
var ctx = cargaContextoCanvas('canvas2');
if(ctx){
ctx.fillStyle = '#990000';
ctx.beginPath();
ctx.moveTo(50,15);
ctx.lineTo(112,15);
ctx.lineTo(143,69);
ctx.lineTo(112,123);
ctx.lineTo(50,123);
ctx.lineTo(19,69);
ctx.closePath();
ctx.fill();
}
```

3.3.3.- 3.- Camino sin relleno y sin cierre

Ahora retomemos los caminos realizados sólo con una línea, sin rellenar de color, que ya vimos en el artículo anterior.

Vendrá bien para ver las diferencias entre los caminos que tienen el color de relleno.

Como veremos, el camino es exactamente igual que los anteriores, con la diferencia que para dibujar sólo la línea del contorno del camino se utiliza el método stroke() del objeto contexto de canvas, en lugar de usar fill() que hace los caminos con relleno de color.

Además, podremos observar como al usar el método stroke() no se cierra automáticamente el camino como ocurría con fill(), sino que se queda abierto.

```
//Hexagono sólo línea, sin cierre de camino
var ctx = cargaContextoCanvas('canvas3');
if(ctx){
ctx.beginPath();
ctx.moveTo(50,15);
ctx.lineTo(112,15);
ctx.lineTo(143,69);
ctx.lineTo(112,123);
ctx.lineTo(50,123);
ctx.lineTo(19,69);
ctx.stroke();
}
```

3.3.4.- 4.- Camino sin relleno y con cierre

Para acabar con estos ejemplos de dibujo de hexágonos en un canvas vamos a mostrar cómo realizar un camino sólo con la línea del borde, como el anterior, pero con el cierre de camino que se consigue con closePath().

El camino es el mismo, pero antes de llamar a stroke() para dibujar la línea, hacemos un closePath() para cerrar el camino.

Para añadir algún interés adicional al camino, hemos utilizado un color distinto para la línea del contorno, que se consigue en esta ocasión con la propiedad strokeStyle del objeto contexto del canvas.

```
//Hexagono sólo línea, con cierre de camino closePath()
var ctx = cargaContextoCanvas('canvas4');
if(ctx){
ctx.strokeStyle = '#990000';
ctx.beginPath();
ctx.moveTo(50,15);
ctx.lineTo(112,15);
ctx.lineTo(143,69);
ctx.lineTo(112,123);
ctx.lineTo(50,123);
ctx.lineTo(19,69);
ctx.closePath();
ctx.stroke();
}
```

Hasta aquí llega esta práctica de caminos en Canvas del HTML 5, con distintas variantes a partir de los mismos puntos del camino.

3.4.- Otros ejemplos de dibujo de caminos en elementos Canvas

A lo largo del Manual del componente Canvas del HTML 5 hemos visto varios ejemplos de dibujo de caminos. Si hemos seguido los capítulos anteriores de este manual ya deberíamos haber cogido un poco de práctica al dibujar líneas que siguen un camino, relleno de color o sin relleno. Lo cierto es que ya podríamos pasar a algún otro tema más adelantado, pero tenemos todavía en el tintero un par de ejemplos con caminos que pueden ser interesantes para acabar de entender cómo se crean.

En el presente ejemplo estamos haciendo varios caminos en un mismo canvas y además, vamos a rellenar de colores distintos cada uno de los caminos, lo que nos vendrá bien para seguir practicando. La idea de este artículo es que nos podamos familiarizar un poco más con la práctica de abrir caminos, cerrarlos y volver a abrir otros caminos. Además, podemos ver que con un mismo camino también podemos pintar en dos partes distintas del lienzo, trasladando el puntero de dibujo pero sin pintar.

3.4.1.- Primer ejemplo, pintar dos caminos distintos

Este primer ejemplo tendría el siguiente código:

```
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
//primer camino
ctx.beginPath();
ctx.moveTo(20,10);
ctx.lineTo(32,20);
ctx.lineTo(22,20);
ctx.lineTo(22,35);
ctx.lineTo(17,35);
ctx.lineTo(17,20);
ctx.lineTo(7,20);
//ctx.closePath(); opcional antes de un fill()
ctx.fill();
//creo un segundo camino
ctx.beginPath(); //probar a comentar esta línea para ver lo que pasa
ctx.fillStyle = '#ff8800';
```

```
ctx.moveTo(47,50);
ctx.lineTo(67,70);
ctx.lineTo(67,30);
ctx.closePath();
ctx.fill();
}
```

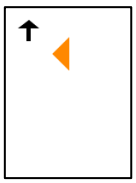
Nota: Lo cierto es que este código está incompleto, pues le falta la función cargaContextoCanvas() que ya se ha explicado anteriormente en el artículo Entender el lienzo de canvas.

En ese código estamos realizando dos caminos distintos sobre un mismo canvas. El primer camino está separado en el código del segundo y los dos comienzan con un beginPath(). En cada camino hacemos un moveTo() para colocar el puntero de dibujo en las coordenadas deseadas.

Luego se hace el closePath() para cerrar el camino, completándolo con una línea recta desde el último punto hasta el punto desde donde comenzamos el caminio. Pero como se puede ver en ejemplo, la llamada al método closePath() es opcional, pues estos dos caminos se rellenan de color con fill() y este método requiere que el camino esté cerrado. Por eso, si el camino no se cerró explícitamente con closePath(), con la llamada a fill() se hace implícitamente.

Otra cosa interesante es el cambio de color que hacemos en el segundo camino con la propiedad fillStyle del objeto canvas, en la línea:

```
ctx.fillStyle = '#ff8800';
```



3.4.2.- Segundo ejemplo, un camino que pinta en dos lugares distintos

El segundo ejemplo que nos queda por ver es muy parecido al primero, con la diferencia que ahora vamos a dibujar la silueta o contorno, en vez de rellenarlos de color. Además, en este segundo ejemplo sólo tenemos un camino en vez de dos que

había antes. Esto es porque hacemos sólo un beginPath() y aunque cerremos el camino con closePath() y luego hagamos un moveTo() para trasladar el puntero de dibujo, en realidad sólo tenemos un camino.

Es por ello que, el cambio de color con la propiedad strokeStyle, aunque se haga en el medio del código, afecta a todo el trazado, pues es el mismo camino.

```
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
ctx.beginPath();
ctx.moveTo(20,7);
ctx.lineTo(32,20);
ctx.lineTo(22,20);
ctx.lineTo(22,35);
ctx.lineTo(17,35);
ctx.lineTo(17,20);
ctx.lineTo(7,20);
ctx.closePath();
//cambio el color de la línea, el color cambia para todo el trazo
ctx.strokeStyle = '#ff8800';
//sigo en el mismo camino, pero muevo el puntero de dibujo
ctx.moveTo(47,50);
```

```
ctx.lineTo(67,70);
ctx.lineTo(67,30);
ctx.closePath();
ctx.stroke();
}
```

Hemos de admitir que estos dos ejemplos no significan un claro avance con respecto a lo que ya habíamos relatado en el manual, pero nunca está de más hacer ejemplos prácticos. Además, hay muchas cosas que merece la pena practicar para entender bien cómo se realizan. En el siguiente artículo explicaremos nuevas técnicas para hacer líneas curvas y no sólo líneas rectas como hasta ahora.

3.5.- Curvas en caminos de Canvas del HTML 5

Los caminos en los elementos canvas del HTML 5 pueden tener curvas, que conseguimos por medio de los arcos, las curvas cuadráticas y las curvas bezier.

Hasta ahora en el manual de canvas del HTML 5 hemos aprendido a hacer caminos con líneas rectas, así que vamos a avanzar un poco más en la materia aprendiendo a dibujar caminos con curvas.

En principio las librerías de funciones para dibujo en el canvas permite tres métodos de para hacer trazos en curva, basados en funciones matemáticas para expresar curvas de distintos tipos:

Arcos:

Nos permiten dibujar circunferencias o segmentos de circunferencias, lo que se conoce como arcos. Lo conseguimos con el método arc() enviando una serie de parámetros que veremos más adelante.

Curvas cuadráticas:

Es una manera de especificar una curva en la que tenemos un punto de inicio, un punto de fin y un tercer punto que indica hacia qué parte se curvará la línea. Esta curva veremos que es fácil de entender y que nos servirá para hacer esquinas redondeadas, entre otras muchas cosas.

Curvas Bezier:

Es una manera matemática de expresar una curva por medio de cuatro puntos. El punto de inicio, el de fin y dos puntos que indicarán hacia dónde se curvará la primera y segunda mitad de la línea. Es una curva un poco más compleja de entender,

pero posiblemente ya hayamos experimentado con este tipo de curvas en programas de diseño como Photoshop o Ilustrator, lo que podrá ayudar un poco a comprenderla.

La verdad es que para hacer todas estas curvas hay que saber un poco de matemáticas y habría que hacer cálculos para poder ajustarlas a nuestras necesidades. Digamos que todas las fórmulas están pensadas para el dibujo técnico y no artístico, por eso quizás un matemático tendría más soltura que un artista para dibujar cosas en el lienzo del canvas.

No obstante, no podemos dejar que pensar que el dibujo en canvas es un proceso informatizado y como estamos diseñando a nivel de lenguaje de programación, no queda otra cosa que adaptarse a las fórmulas matemáticas implementadas para hacer curvas. Más adelante veremos otras maneras de solventar estos temas, como la utilización de imágenes, que podemos importar y "pegar" en el canvas, a partir de archivos gráficos creados con cualquier programa como Photoshop.

En los siguientes artículos veremos con detalle cada uno de estos tres tipos de curvas con sus ejemplos. Podemos comenzar conociendo las curvas en arcos.

3.6.- Dibujo de curvas con arcos en canvas

Cómo dibujar arcos, para hacer curvas basadas en circunferencias o segmentos de ellas, en el lienzo de los elementos Canvas del HTML 5.

En este artículo explicaremos cómo hacer caminos con arcos. Los arcos son segmentos de circunferencias, o una circunferencia entera, en el caso de un arco completo. Son uno de los modos de hacer curvas en el elemento Canvas del HTML 5.

El método que podemos dibujar para hacer un arco es arc(), que invocamos sobre el objeto el contexto del canvas. Este método requiere unos cuantos parámetros para poder invocarlo y especificar las características del arco que se desea hacer y lo cierto es que no resulta del todo trivial porque hay que conocer algunas fórmulas

matemáticas para el trabajo con circunferencias. Así que tendremos que refrescar algunos conocimientos que pueden haberse olvidado del periodo de enseñanza media.

Nota: Igual que los caminos, una vez creados, podemos decidir si queremos rellenarlos de color, mediante el método fill() o bien dibujar solamente el contorno, con el método stroke().

Estos son los parámetros que debemos enviar al método arc():

```
arc(x, y, radio, angulo_inicio, angulo_final, sentido_contrario_del_reloj)
```

• Los parámetros x, y corresponden con las coordenadas del centro del arco.

• El parámetro radio es el número de píxeles que tiene el arco como radio.

• Por su parte angulo_inicio y angulo_final son los ángulos donde comienza y acaba el radio. Están tomados como si

el eje de la horizontal tuviese el ángulo cero.

• Sentido_contrario_del_reloj es un parámetro boleano, donde true significa que el trazo va desde un ángulo de inicio al de fin en el sentido contrario de las agujas del reloj. False indica que ese camino es en dirección contraria.

La verdad es que todos los parámetros son bastante sencillos de entender, pero el ángulo de inicio y fin no se indican en grados, como podríamos suponer, sino en radianes. Para el que no se acuerde, se puede hacer un paso de grados a radianes atendiendo a la siguiente fórmula:

```
Radianes = número_PI x (grados/180)
```

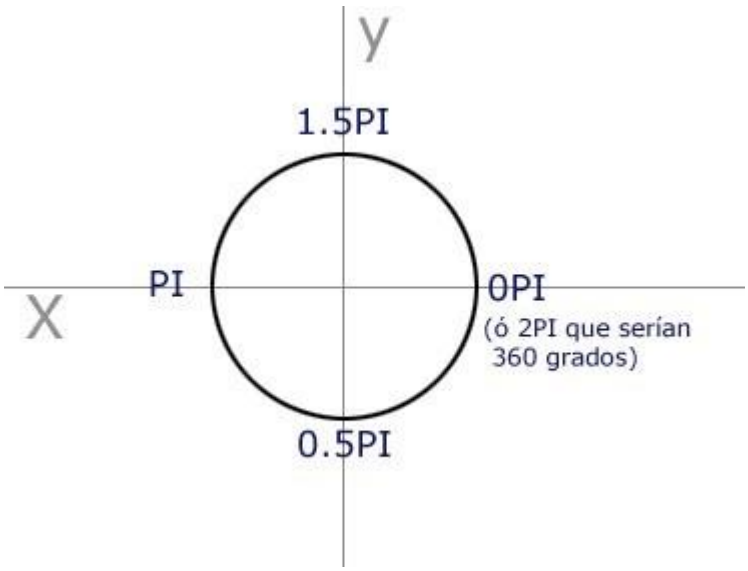
Para convertir grados en radianes podríamos utilizar la siguiente línea de código Javascript:

```
var radians = (Math.PI/180)*degrees
```

Nota: Math.PI es el famoso número PI (3.1416). En Javascript, a partir de la clase Math, tenemos acceso a esa constante, así como diversas funcines matemáticas. Ver las notas sobre la clase Math.

3.6.1.- Entender los radianes

Para comprender los gradianes de una manera más visual, así como la referencia sobre el eje X, que serían los cero grados, se puede ver la siguiente imagen:



En la imagen anterior tenemos varios valores de radianes:

• 0 Radianes serían cero grados y es el punto marcado por 0PI, en el eje de las X y a la derecha del centro de la circunferencia.

• 0.5 PI Radianes serían 90 grados el punto del eje de las Y abajo del centro.

• 1 PI Radianes es media circunferencia, 180 grados.

• 1.5 PI Radianes sería el equivalente a 270 grados

• 2 PI Radianes son los 360 grados de la circunferencia completa y correspondería con el mismo punto que los cero grados.

Así pues, para hacer un círculo completo con centro en (50, 50) de 20 píxeles de radio, podríamos utilizar un código como este:

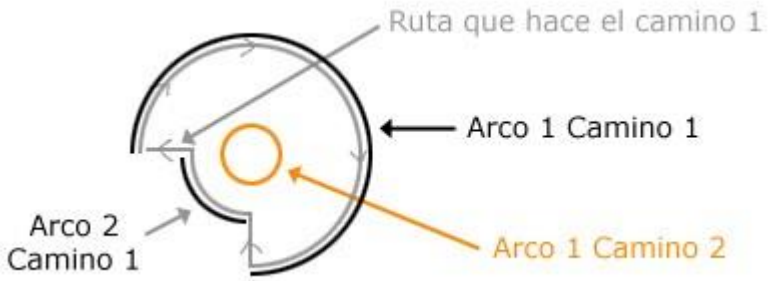
```
contextoCanvas.arc(50, 50 ,20 , 0, Math.PI*2, false);
```

Como se puede ver, la circunferencia empieza en 0 PI (cero) y termina en 2 PI.

3.6.2.- Ejemplo de dibujo de caminos con arcos

Para que se pueda entender el método con el que se crean caminos complejos a base de arcos en el elemento Canvas, vamos a presentar el siguiente ejemplo, en el que crearemos este sencillo diseño.

En realidad aunque parezca una figura un poco compleja de hacer, se consigue con dos caminos. El primer camino se rellena con color negro y el segundo con color naranja. En la siguiente imagen se puede ver de una manera más clara los caminos que habría en para hacer ese diseño.



El primer camino tiene dos arcos concéntricos, uno con radio mayor y el segundo con un radio menor. Este primer camino comienza en el radio mayor y se puede ver una línea gris que hemos puesto, con unas flechas, para poder reconocer la dirección que lleva el camino.

3.6.3.- Veamos el código para hacer este diseño.

```
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
//primer camino, en negro
ctx.beginPath();
ctx.arc(75,75,60,Math.PI,Math.PI*0.5,false);
ctx.arc(75,75,32,Math.PI*0.5,Math.PI*1,false);
ctx.closePath()
ctx.fill();
//segundo camino, en naranja
ctx.fillStyle = '#ff8800';
ctx.beginPath();
ctx.arc(75,75,15,0,Math.PI*2,false);
ctx.fill();
}
```

Recordar que este código es parcial, puesto que las partes que faltarían para completarlo, como la función

cargaContextoCanvas() o el HTML del elemento canvas, ya las conocemos de diversos artículos anteriores del Manual de Canvas.

3.7.- Curvas cuadráticas en el canvas

Las curvas cuadráticas son un tipo especial de curva que se define por tres puntos, dos para el inicio y fin de la curva y otro para su tendencia.

En un articulo anterior del manual de Canvas del HTML 5 ya explicamos los tipos de curvas que podemos definir al dibujar en el lienzo. Recordemos que para expresar cualquier dibujo en un canvas necesitamos realizar sentencias en lenguajes de programación, que sólo nos permiten dibujar por medio de la definición de parámetros matemáticos, por lo que a veces el dibujo puede ser una tarea más complicada que coger un lapiz y pintar sobre papel.

En este caso vamos a revisar un tipo de curva llamada Cuadrática, que nos sirve bien para hacer curvas sencillas, no necesariamente arcos de una circunferencia, con un único punto de inflexión. Por intentar explicarlo con palabras de manera entendible, podríamos decir que las curvas cuadráticas permiten expresar una única curvatura entre dos puntos. Para expresarlas tenemos un punto inicial, un punto final de la curva y un punto que define la tendencia de la curvatura.

Las curvas cuadráticas son un tipo concreto de curvas Bezier, es decir, una manera de expresar matemáticamente una curva, similar a las Bezier pero más simplificada. Mientras que en las curvas Bezier tenemos dos puntos para definir la tendencia de la curva, al principio y el fin de la misma, en las curvas cuadráticas sólo tendremos un punto.

Nota: No hemos visto todavía las mencionadas curvas Bezier, pues son más complejas que las curvas cuadráticas. Es por eso que las veremos más adelante.

3.7.1.- Método quadraticCurveTo() para dibujar curvas cuadráticas

Las curvas cuadráticas actúan como otros métodos para dibujar caminos en el canvas. Recordemos que al hacer un camino en el canvas partimos de un punto inicial, que es el punto donde está situado el puntero de dibujo (podríamos imaginar ese punto inicial como el lugar donde está situado el lápiz antes de empezar a dibujar la curva). Así que, para expresar una curva cuadrática, tendremos que definir el punto final de la misma y el punto imaginario hacia el que se curvará la línea entre esos dos puntos.

Utilizaríamos la siguiente llamada a un método del contexto del canvas.

```
quadraticCurveTo(pcx, pcy, x, y)
```

Este método recibe cuatro valores, que corresponden con dos puntos del lienzo. Insisto en que el punto inicial ya está implícito en el contexto del canvas, con la posición dada del puntero de dibujo antes de comenzar la curva cuadrática.

Luego, el punto (pcx, pcy) es el lugar "imaginario" al que tendrería la curvatura de la línea. El punto (x,y) sería el final de la curva.

Una manera sencilla de entender este método sería ver la siguiente imagen:

En el anterior gráfico tenemos tres puntos:

1. El primero, marcado con color morado, es la posición del puntero de dibujo al iniciar la curva cuadrática. Ese punto no lo definimos al hacer la llamada al método quadraticCurveTo() porque ya está implícito en el contexto del canvas. En cualquier caso se puede cambiar con una llamada a moveTo() como hemos visto en artículos anteriores.

2. El segundo punto, marcado con color rojo, es la tendencia de la curva cuadrática. Ese punto decimos que es

imaginario porque no aparece en la curva. Simplemente sirve para definir cómo será la curvatura. Se define con los parámetros pcx, pcy.

3. El tercer punto, dibujado en verde, es el final de la curva, definido por los parámetros x, y.

3.7.2.- Ejemplo de curva cuadrática

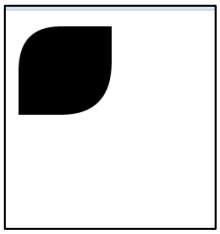
Ahora podemos ver un ejemplo de dibujo en canvas de un camino que incluye dos curvas cuadráticas.

```
var ctx = cargaContextoCanvas('micanvas');

if(ctx){
```

```
ctx.beginPath();
ctx.moveTo(10,60)
ctx.quadraticCurveTo(10,10,60,10);
ctx.lineTo(120,10);
ctx.lineTo(120,50);
ctx.quadraticCurveTo(120,110,60,110);
ctx.lineTo(10,110);
ctx.fill();
}
```

Como se puede ver, aparte de la curva cuadrática tenemos otras líneas rectas en este dibujo, que luego se rellena de color con fill(). Nos da el resultado una forma parecida a una hoja, que se puede ver en el ejemplo:



.

3.8.- Rectángulos con esquinas redondeadas en canvas, interactivo con Mootools

Vemos las curvas cuadráticas a través de un ejemplo interactivo de trabajo con caminos en canvas del HTML 5 que dibuja rectángulos con esquinas redondeadas.

Vamos a mostrar un nuevo ejemplo de dibujo de caminos en canvas un poco más avanzado. Crearemos una página con un canvas que tendrá un rectángulo con esquinas redondeadas y una interfaz de usuario para que se pueda configurar el radio del redondeado de las esquinas.

Es un ejemplo un poco avanzado porque mezclamos varias tecnologías, pues no sólo tenemos que pintar en el canvas, sino también responder a acciones del usuario para alterar el dibujo.

Por un lado tenemos que saber hacer dibujos en canvas con curvas cuadráticas. De hecho, este ejemplo de trabajo en canvas del HTML 5 nos ayudará a observar un poco más la utilidad de las curvas cuadráticas.

Para que el usuario pueda definir el radio de las curvas en las esquinas del rectángulo vamos a colocar una interfaz de tipo "slider" creada con el framework Javascript Mootools, que permite cambiar el valor del radio arrastrando un control.

Además habrá un campo de texto para cambiar este radio escribiendo cualquier otro valor directamente.

Para saber mejor qué es lo que vamos a crear, recomendamos echar un vistazo a la página del ejemplo.

3.8.1.- Función para crear un rectángulo con esquinas redondeadas en canvas

En las páginas de ayuda para trabajar con caminos del canvas de Mozilla hay un código de una función para hacer rectángulos con esquinas redondeadas que vamos a utilizar para este ejemplo.

```
function roundedRect(ctx,x,y,width,height,radius){
ctx.beginPath();
ctx.moveTo(x,y+radius);
ctx.lineTo(x,y+height-radius);
ctx.quadraticCurveTo(x,y+height,x+radius,y+height);
ctx.lineTo(x+width-radius,y+height);
ctx.quadraticCurveTo(x+width,y+height,x+width,y+height-radius);
```

```
ctx.lineTo(x+width,y+radius);
ctx.quadraticCurveTo(x+width,y,x+width-radius,y);
ctx.lineTo(x+radius,y);
ctx.quadraticCurveTo(x,y,x,y+radius);
ctx.stroke();
}
```

Simplemente hace un rectángulo en la posición x,y con anchura y altura dadas por medio de los parámetros width y height y un último parámetro radius para especificar el radio de la curvatura en la esquina redondeadas.

Ahora podríamos hacer un rectángulo redondeado con la siguiente llamada:

```
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
roundedRect(ctx, 10, 10, 130, 110, 20);
}
```

3.8.2.- Javascript para cambiar el radio de las esquinas redondeadas

Ahora veamos el código Javascript para alterar el radio de las esquinas como respuesta a eventos del usuario. Primero observemos esta función Javascript, que recibe un valor radio y sirve para actualizar el rectángulo del canvas:

```
function actualizaRadioRectangulo(radio){
radio = parseInt(radio)
if (isNaN(radio)) {
radio = 0;
}
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
ctx.clearRect(0,0,150,150);
roundedRect(ctx, 10, 10, 130, 110, radio);
}
}
```

Ahora podemos ver el campo de texto para cambiar el radio de las esquinas manualmente, escribiendo cualquier otro valor dentro del mismo.

```
Radio: <input type="text" name="radio" value="10" onkeyup="actualizaRadioRectangulo(this.value)">
```

Como se puede ver, tiene definido un evento para actualizar el radio del rectángulo cuando el usuario pulsa una tecla en el campo de texto.

3.8.3.- Código Mootools para el componente slider

Hasta el momento no se había utilizado para nada Mootools, es decir, todo lo que hemos visto es Javascript normal. Para lo que necesitamos el framework Javascript es para el componente slider, que es una interfaz de usuario para cambiar valores al mover una barra que se desplaza a izquierda o derecha.

Ese componente slider está en la distribución Mootools que se llama "more" y tenemos que descargarla por separado en la propia página de descarga de Mootools, accediendo mediante el enlace que pone "More Builder". Allí tenemos que seleccionar por lo menos el componente "Slider" y los paquetes requeridos se seleccionarán automáticamente.

Nota: Recordemos que el "More" de Mootools son una serie de scripts para crear interfaces de usuario avanzadas. Se descarga por separado del "Core", que es el framework fundamental. Por supuesto, para poder implementar los componentes del "More" se necesita tener disponible el "Core". En principio dicen en la página de Mootools que para ejecutar cualquier componente del "More" es necesario haber descargado el "Core" completo.

Así pues, para la parte del slider tenemos que incluir los scrips "Core" y "More"

```
<script src="mootools-1.2.4-core-yc.js" type="text/javascript"></script>
```

```
<script src="mootools-1.2.4.2-more.js" type="text/javascript"></script>
```

Luego podríamos tener un HTML como este para producir el contenedor del slider:

```
<div id="slidercontenedor" style="width:220px; padding: 5px 0px; background-color:#eeeeee;">
<div id="slidercontrol" style="width:10px; height: 10px; background-color:#9999dd;"></div>
</div>
<div>Valor: <span id="valor">20</span></div>
```

Ahora podemos ver el script Mootols para generar dinámicamente el componente a partir de estos elementos HTML.

```
window.addEvent("domready", function(){
var miSlider = new Slider("slidercontenedor", "slidercontrol",{
'range': [0,55],
'steps': 55,
'initialStep': 20,
onChange: function(lugar){
actualizaRadioRectangulo(lugar);
$("valor").set("html", lugar);
}
});
});
```

3.8.4.- Código completo del ejercicio

Para acabar este ejercicio nos quedan algunas cosas que no hemos comentado sobre el elemento canvas del HTML 5, porque se habían visto anteriormente en repetidos artículos del Manual de Canvas, como la función cargaContextoCanvas()

De todos modos, para referencia podemos ver a continuación el código completo de este creador dinámico e interactivo de rectángulos redondeados.

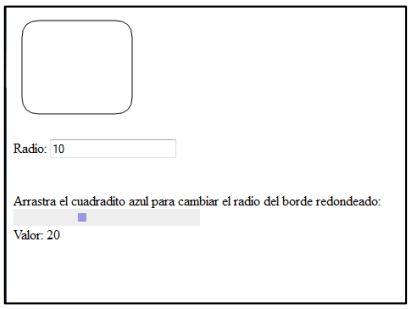
```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<script src="mootools-1.2.4-core-yc.js" type="text/javascript"></script>
<script src="mootools-1.2.4.2-more.js" type="text/javascript"></script>
<title>Curvas cuadráticas</title>
<script>
//Recibe un identificador del elemento canvas y carga el canvas
//Devueve el contexto del canvas o FALSE si no ha podido conseguise
function cargaContextoCanvas(idCanvas){
var elemento = document.getElementById(idCanvas);
if(elemento && elemento.getContext){
var contexto = elemento.getContext('2d');
if(contexto){
return contexto;
}
}
return FALSE;
```

```
}
//Crea un rectángulo con las esquinas rendondeadas
function roundedRect(ctx,x,y,width,height,radius){
ctx.beginPath();
ctx.moveTo(x,y+radius);
ctx.lineTo(x,y+height-radius);
ctx.quadraticCurveTo(x,y+height,x+radius,y+height);
ctx.lineTo(x+width-radius,y+height);
ctx.quadraticCurveTo(x+width,y+height,x+width,y+height-radius);
ctx.lineTo(x+width,y+radius);
ctx.quadraticCurveTo(x+width,y,x+width-radius,y);
ctx.lineTo(x+radius,y);
ctx.quadraticCurveTo(x,y,x,y+radius);
ctx.stroke();
}
function actualizaRadioRectangulo(radio){
radio = parseInt(radio)
if (isNaN(radio)) {
radio = 0;
}
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
ctx.clearRect(0,0,150,150);
roundedRect(ctx, 10, 10, 130, 110, radio);
}
}
window.onload = function(){
//Recibimos el elemento canvas
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
roundedRect(ctx, 10, 10, 130, 110, 20);
}
}
</script>
```

```
<script>
window.addEvent("domready", function(){
var miSlider = new Slider("slidercontenedor", "slidercontrol",{
'range': [0,55],
'steps': 55,
'initialStep': 20,
onChange: function(lugar){
actualizaRadioRectangulo(lugar);
$("valor").set("html", lugar);
}
});
});
```

```
</script>
</head>
<body>
<canvas id="micanvas" width="150" height="150">
Recicla tu navegador!!
</canvas>
<form name="fradio">
Radio: <input type="text" name="radio" value="10" onkeyup="actualizaRadioRectangulo(this.value)">
</form>
<br><br>
Arrastra el cuadradito azul para cambiar el radio del borde redondeado:
<div id="slidercontenedor" style="width:220px; padding: 5px 0px; background-color:#eeeeee;">
<div id="slidercontrol" style="width:10px; height: 10px; background-color:#9999dd;"></div>
</div>
<div>Valor: <span id="valor">20</span></div>
</p>
</body>
</html>
```

VER:



3.9.- Curvas Bezier en Canvas

Las curvas Bezier son la manera más compleja de especificar dibujar caminos curvos en el elemento canvas del HTML 5.

Bezier es el último de los tipos de curva sobre caminos en elementos canvas que nos queda por ver en el Manual de trabajo con el canvas del HTML 5.

El modelo que propone Bezier es un tipo de función matemática para definir curvas complejas en función de varios valores.

Es una técnica utilizada en el dibujo técnico, que surgió inicialmente en el mundo de la aeronáutica y el diseño de coches y que se hizo bastante popular a raíz de su utilización en varios programas de diseño, como el conocido Photoshop. Las curvas Bezier se crean por medio de una fórmula matemática que permite especificar y evaluar trazados curvos que podrían tener más de un punto de inflexión.

3.9.1.- Método para dibujar curvas Bezier

En el dibujo con el elemento Canvas se han implementado las curvas Bezier a partir del siguiente método del contexto del canvas.

```
bezierCurveTo(pc1x, pc1y, pc2x, pc2y, x, y)
```

Como vemos, se tienen que especificar coordenadas de tres puntos, de una manera similar a la que conocimos en las curvas cuadráticas.

Nota: Las curvas cuadráticas un tipo determinado de curvas Bezier, lo que ocurre es que en las curvas Bezier utilizamos dos puntos de tendencia de la curva, para el principio y el final de la misma, mientras que en las curvas cuadráticas sólo se utilizaba uno. Para aclarar este

punto recomendamos echar un vistazo a las explicaciones sobre curvas cuadráticas.

En la siguiente imagen se puede ver un diagrama sobre los puntos que se utilizan para definir una curva Bezier.

Como podemos ver, el método bezierCurveTo() tiene 6 parámetros que corresponden con las coordenadas de 3 puntos, pero en la imagen se utilizan hasta 4 puntos para definir la curva Bezier, pues el punto de incicio de la curva ya estaba en el contexto del canvas. Así que, atendiendo a la anterior imagen, estos serían los puntos necesarios para componer la curva Bezier:

1. El primer punto, marcado con color morado, es el punto inicial de la curva. Este punto no se tiene que definir, pues ya está implícito en el contexto del canvas, en el lugar donde estaba el puntero de dibujo al llamar al método bezierCurveTo().

Nota: Al dibujar cualquier segmento de un camino tenemos definido siempre de antemano el punto inicial de ese segmento del camino, pues es el lugar donde está el puntero de dibujo. Nosotros podríamos cambiar ese puntero de dibujo, para cambiar el primer punto de la curva, con una llamada al método moveTo().

2. El segundo punto, que se ha marcado de color verde, es la tendencia de la primera parte de la curva, que se indica con los parámetros pc1x, pc1y.

3. El tercero, marcado de color rojo, es la tendencia de la segunda parte de la curva, que se indica con los parámetros pc2x, pc2y.

4. Finalmente, tenemos el punto final de la curva, marcado en color gris, que se indica con los parámetros x,y.

3.9.2.- Ejemplo de dibujo con curvas Bezier

Ahora podemos crear un ejemplo para que los lectores puedan terminar de entender las curvas Bezier. Veamos un camino que contiene alguna recta y varias curvas Bezier.

```
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
ctx.beginPath();
ctx.fillStyle = "#ccccff";
ctx.moveTo(0,40);
ctx.bezierCurveTo(75,17,70,25,100,60);
ctx.bezierCurveTo(130,35,140,45,145,50);
ctx.bezierCurveTo(180,45,190,55,200,70);
ctx.lineTo(200,150);
ctx.lineTo(0,150);
ctx.fill();
}
```

Ahora podemos complicar un poco más ese ejemplo para crear otros caminos con curvas Bezier, con la particularidad de que vamos a rellenarlos con colores semitransparentes.

Nota: Nosotros asignamos colores de relleno para los caminos con la propiedad fillStyle del objeto contexto del canvas. Podemos asignar un color con un código RGB de una manera que ya conocemos:

```
ctx.fillStyle = "#ccccff";
```

Pero aparte, también podemos indicar colores con valores RGB en decimal, de manera similar a como se hace en CSS, e incluso podemos asignar valores RGBA (con canal alpha para la transparencia).

```
ctx.fillStyle = 'rgba(100,230,100,0.3)';
ctx.beginPath();
ctx.fillStyle = 'rgba(100,230,100,0.3)';
ctx.moveTo(0,90);
ctx.bezierCurveTo(90,7,110,15,140,30);
ctx.bezierCurveTo(130,55,140,65,145,70);
ctx.bezierCurveTo(180,45,190,55,200,95);
ctx.lineTo(200,150);
```

```
ctx.lineTo(0,150);
ctx.fill();
ctx.beginPath();
ctx.fillStyle = 'rgba(230,230,100,0.3)';
ctx.moveTo(50,150);
ctx.bezierCurveTo(90,7,110,15,160,10);
ctx.bezierCurveTo(130,105,140,135,200,35);
ctx.lineTo(200,150);
ctx.lineTo(0,150);
ctx.fill();
```

Parte 4:

Trabajo con imágenes en Canvas

A través de la utilización de imágenes se pueden mejorar mucho los diseños que se realizan en los canvas. Además mostramos distintas maneras de modificar dinámicamente por medio de scripts el aspecto de las imágenes.

4.1.- Usar imágenes en el Canvas

Dibujar el contenido de imágenes en los elementos canvas del HTML 5, usando cualquier tipo de archivo gráfico permitido (gif, jpg, png) para incluir una imagen en el lienzo de un canvas.

Una de las cosas más interesantes que podremos hacer cuando dibujamos en el lienzo del elemento canvas es importar y mostrar directamente el contenido de archivos gráficos externos, es decir, usar imágenes GIF, JPG o PNG dentro de los dibujos que realizamos con canvas. En este artículo veremos cómo realizar este punto, aunque adelantamos que es bastante fácil.

Las imágenes provenientes de archivos gráficos las podemos crear con nuestro editor preferido y hacer fácilmente gráficos bastante creativos y vistosos, o editar a partir de fotos creadas con nuestra cámara. Luego las podemos incluir en el Canvas y así conseguir que nuestros trabajos tengan una mejor calidad que si dibujamos a mano con las funciones Javascript del API de Canvas.

Con un poco de creatividad y algo de código Javascript, podremos hacer composiciones basadas en varias imágenes "pegadas" en el lienzo, o utilizar imágenes de fondo sobre las que luego pintamos con Javascript para destacar cosas. Como podemos usar cualquier tipo de archivo gráfico, mientras que esté soportado por el navegador, las posibilidades son enormes.

4.1.1.- Método drawImage() para pintar una imagen en el canvas

Para dibujar una imagen en el lienzo se utiliza el Método drawImage(), que pertenece al objeto contexto del canvas, con la siguiente sintaxis:

```
drawImage(objeto_imagen, x, y)
```

Enviamos tres parámetros, el primero es el objeto Javascript de la imagen que se desea incluir en el lienzo. Los dos siguientes son las coordenadas donde situar la imagen, siendo (x,y) el punto donde se colocará la esquina superior izquierda de la imagen.

Como decía, este método pertenece al objeto del canvas, por lo que antes de poder invocarlo debemos haber obtenido el contexto del canvas, tal como hemos aprendido anteriormente en el Manual de Canvas para cualquier otro tipo de dibujo.

4.1.2.- Objeto Javascript imagen

El objeto imagen es uno de los objetos básicos de Javascript, que afortunadamente funciona igual en todos los navegadores.

Este objeto de imagen lo podemos obtener de varias maneras, pero de momento vamos a aprender a generarlo

dinámicamente con una instrucción Javascript.

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

Con esto tenemos una variable llamada "img" que tiene un objeto imagen dentro. Ese objeto imagen en estos momentos está sin ningún atributo. Podríamos decir que está sin inicializar. La tarea de inicialización fundamental sería asignarle una ruta a un archivo gráfico.

```
img.src = 'logo-grande.jpg';
```

Esto hace que en el objeto Image se cargue la imagen que está en el archivo 'logo-grande.jpg' y como no hemos especificado ningún directorio en la ruta, se supone que ese archivo está en la misma carpeta que el archivo HTML donde esté ese código Javascript.

Una vez tenemos el objeto imagen, podríamos pintarlo en un canvas por medio de la función drawImage(). Sería algo parecido a esto:

```
ctx.drawImage(img, 10, 10);
```

Pero atención, porque este código tiene un detalle: la imagen no se dibujará en el canvas a no ser que esté previamente cargada en el navegador.

En la secuencia de instrucciones, tal como lo tenemos ahora:

```
var img = new Image();
img.src = 'logo-grande.jpg';
ctx.drawImage(img, 10, 10);
```

El navegador al especificar el archivo de la imagen, actualizando el atributo src, tiene que descargarlo y eso lleva un tiempo.

Por tanto, si inmediatamente a indicar el archivo, intentamos dibujar la imagen, dará un problema. Dicho de otra manera, sólo podemos dibujar la imagen cuando estamos seguros que el navegador ya la ha descargado. Para asegurarnos de este punto, podemos usar el evento onload de la imagen, para llamar a drawImage() sólo cuando la imagen ha terminado de cargarse.

```
var img = new Image();
img.src = 'canvas-html5.png';
img.onload = function(){
ctx.drawImage(img, 10, 10);
}
```

4.1.3.- Ejemplo de dibujo de imagen en un canvas

Veremos a continuación el código completo de un ejemplo que carga una imagen en un elemento canvas.

```
<html>
<head>
<title>Imágenes en Canvas</title>
<script language="javascript">
function cargaContextoCanvas(idCanvas){
var elemento = document.getElementById(idCanvas);
if(elemento && elemento.getContext){
var contexto = elemento.getContext('2d');
if(contexto){
return contexto;
}
```

```
}
return FALSE;
}
window.onload = function(){
//Recibimos el elemento canvas
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
//Creo una imagen conun objeto Image de Javascript
var img = new Image();
//indico la URL de la imagen
img.src = 'logo-    .gif';
//defino el evento onload del objeto imagen
img.onload = function(){
//incluyo la imagen en el canvas
ctx.drawImage(img, 10, 10);
}
}
}
</script>
</head>
<body>
<canvas id="micanvas" width="200" height="100">
Tu navegador no soporta canvas.
</canvas>
</body>
</html>
```

En el siguiente artículo vamos a mostrar diversas maneras de acceder a objetos Image desde Javascript para mostrar esas imágenes en el canvas.

4.2.- Maneras de acceder a objetos Image para mostrar en el canvas

Otro ejemplo de dibujo en con el API de canvas del HTML 5, en el que incluimos imágenes traídas por diferentes vías posibles con Javascript.

En el artículo anterior explicamos las generalidades del trabajo con imagenes en el elemento canvas del HTML 5.

Continuando con esas explicaciones veremos ahora cómo pintar en un lienzo diversas imágenes que extraemos de varios modos.

La idea es experimentar con el dibujo en Canvas por medio de una nueva práctica y a la vez repasar todos los modos que existen de obtener una imagen por medio de Javascript, que ya explicamos en el artículo Distintas maneras de acceder a objetos Image Javascript.

Como ya se explicó en el mencionado artículo, existen diversas maneras de conseguir objetos Image en Javascript, que luego podríamos pintar en un canvas. Lo iremos viendo directamente sobre el código fuente de este ejemplo:

1.- Traerse una imagen que hay en la página: por medio del método getElementById(), enviando como parámetro el identificador de la etiqueta IMG de la imagen deseada.

```
//Creo un objeto Image con una imagen de la pagina
var img = document.getElementById("im1");
```

```
//luego la dibujo en el canvas
ctx.drawImage(img, 10, 10);
```

2.- A través del Array de images: También de una imagen que haya en la página, en una etiqueta IMG. Al array accedemos con el índice de la imagen según orden de aparición en el código HTML.

```
//consigo una imagen desde el array de imágenes
ctx.drawImage(document.images[1], 122, 20);
```

3.- Creando nuestro objeto Image: Que es la forma con la que trabajamos en el artículo anterior. Y por tanto no vamos a repetir las explicaciones.

```
//un objeto Image
var imagen = new Image();
imagen.src = "http://www. XXXX .com/images/iconos/user_go.png";
imagen.onload = function(){
ctx.drawImage(imagen, 330, 195);
}
```

4.- Especificar la imagen en formato data:url: que es una cadena de caracteres en formato Base64 que permite especificar elementos como imágenes a partir de código, pero como si esos elementos los adquiriésemos remotamente.

```
//a través de un "data: url"
var img = new Image();
img.src =
'
BYAOw==';
ctx.drawImage(img, 300, 200);
```

5.- Acceder a el diseño dibujado en otro canvas: para mostrar en un canvas el contenido de otro, como si fuera una imagen.

```
//consigo una imagen desde un canvas
var imgCanvas = document.getElementById("canvas2");
ctx.drawImage(imgCanvas, 100, 120);
```

Este quinto y último método permite algunas aplicaciones interesantes, como mostrar un canvas una miniatura de lo que hay en otro canvas.

4.2.1.- Ejemplo completo de trabajo en canvas con imágenes de varias fuentes

Para acabar, podemos ver el código completo de una página que utiliza todos esos métodos para acceder a imágenes y mostrarlas en el canvas.

Se podrá ver que en realidad se crean dos canvas. Uno sólo lo creamos para poder copiarlo en otro canvas.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd"
>
<html>
<head>
<title>Imágenes en Canvas</title>
<script language="javascript">
function cargaContextoCanvas(idCanvas){
var elemento = document.getElementById(idCanvas);
```

```
if(elemento && elemento.getContext){
var contexto = elemento.getContext('2d');
if(contexto){
return contexto;
}
}
return false;
}
window.onload = function(){
//cargo un camino en un canvas, para luego traerlo como imagen
var ctx = cargaContextoCanvas('canvas2');
if(ctx){
ctx.fillStyle = '#990000';
ctx.beginPath();
ctx.moveTo(50,15);
ctx.lineTo(112,15);
ctx.lineTo(143,69);
ctx.lineTo(112,123);
ctx.lineTo(50,123);
ctx.lineTo(19,69);
ctx.closePath();
ctx.fill();
}
//Recibimos el elemento canvas
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
//Creo un objeto Image con una imagen de la pagina
var img = document.getElementById("im1");
ctx.drawImage(img, 10, 10);
//consigo una imagen desde el array de imágenes
ctx.drawImage(document.images[1], 122, 20);
//consigo una imagen desde un canvas
var imgCanvas = document.getElementById("canvas2");
ctx.drawImage(imgCanvas, 100, 120);
//un objeto Image
var imagen = new Image();
imagen.src = "http://www.    .com/images/iconos/user_go.png";
imagen.onload = function(){
ctx.drawImage(imagen, 330, 195);
}
//a través de un "data: url"
var img = new Image();
img.src =
'
BYAOw==';
```

```
ctx.drawImage(img, 300, 200);
}
document.images[1].src =
'
BYAOw==';
}
</script>
</head>
<body>
<h2>Canvas que estoy creando con una serie de imágenes</h2>
<canvas id="micanvas" width="500" height="400">
Tu navegador no soporta canvas.
</canvas>
<p>
<div style="display: none;">
<h2>Cosas que pongo aquí para acceder desde Javascript</h2>
<img src="canvas-html5.png" id="im1">
<img src="logo-grande.jpg">
<p>
<canvas id="canvas2" width="150" height="150">
Recicla tu navegador!!
</canvas>
</div>
</body>
</html>
```

Nota: las imágenes se pueden alterar dinámicamente, para mostrarlas en el canvas con algunos cambios.

4.3.- Escalado y recorte en imágenes en canvas

Escalado y recorte de imágenes en el elemento canvas. Como cambiar el tamaño y recortar las imágenes al dibujarlas en el lienzo de canvas del HTML 5.

En pasados artículos del Manual de Canvas estuvimos viendo cómo incluir imágenes, es decir, como dibujar el contenido de una imagen en el lienzo de un elemento canvas del HTML 5. Seguiremos con las explicaciones en el presente texto, ofreciendo unas notas adicionales sobre el tratamiento de imágenes en Canvas, que nos permitirán redimensionar y recortar las imágenes antes de pintarlas.

El método es bien simple y consiste en invocar al método que dibuja las imágenes, drawImage(), enviando distintos juegos de parámetros. Anteriormente ya habíamos trabajado con este método, que como debemos saber, pertenece al objeto contexto de un canvas. En el pasado lo llamamos simplemente enviándole la imagen y las coordenadas donde había que colocarla. Ahora vamos a ver los otros dos modos de invocarlo, por medio de parámetros adicionales, que nos faltan por conocer. El primero de los modos de invocación permite escalar una imagen y el segundo recortarla y escalarla en un mismo paso.

4.3.1.- Escalar una imagen

Redimensionar una imagen es sencillo. Simplemente tenemos que invocar al método drawImage() enviando además las dimensiones de la imagen que queremos que se dibuje. El navegador escalará la imagen para que tenga las dimensiones que indiquemos y luego la pintará en el canvas.

Las nuevas dimensiones de la imagen a dibujar pueden ser las que deseemos. Pueden incluso no ser proporcionales a las dimesiones actuales, en ese caso el navegador estirará la imagen o la achatará para adaptarla a la anchura y altura que hayamos indicado.

La manera de llamar a este método del contexto del canvas es la siguiente:

```
drawImage(imagen, posX, posY, anchura, altura);
```

Este método dibujará la imagen en la posición definida por las coordenadas (posX, posY) y con la anchura y altura dadas en los últimos dos parámetros.

Así que podemos ver un ejemplo de código escalando la imagen:

```
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
//Creo una imagen con un objeto Image de Javascript
var img = new Image();
//indico la URL de la imagen
img.src = 'logo-    .gif';
//defino el evento onload del objeto imagen
img.onload = function(){
//incluyo la imagen en el canvas escala muy pequeña
ctx.drawImage(img, 0, 0, 50 , 24);
//un poco mayor
ctx.drawImage(img, 70, 10, 80 , 38);
//tamaño natural
ctx.drawImage(img, 160, 20);
}
}
```

Este ejemplo dibuja la misma imagen tres veces, dos de ellas está escalada a distintas dimensiones y la última está a tamaño natural (sin redimensionar).

Puedes ver el ejemplo en funcionamiento en una página aparte.

4.3.2.- Recortar y escalar una imagen

El último modo de invocar al método drawImage() es un poco más complejo, ya que le tenemos que indicar todos los datos para poder recortar y escalar la imagen antes de dibujarla en el canvas. La llamada tendrá estos parámetros:

```
drawImage(imagen, imgX, imgY, imgAncho, imgAlto, lienzoX, lienzoY, LienzoAncho, LienzoAlto)
```

Entre los parámetros, "imagen" sigue siendo el objeto imagen Javascript que queremos pintar. Todos los parámetros siguientes los podemos entender a la vista de la siguiente imagen:

Podemos ver a continuación el código de un ejemplo que realiza el recorte y escalado de una imagen.

```
var ctx = cargaContextoCanvas('micanvas');
if(ctx){
//Creo una imagen conun objeto Image de Javascript
var img = new Image();
//indico la URL de la imagen
img.src = 'sagrada-familia.jpg';
//defino el evento onload del objeto imagen
img.onload = function(){
ctx.drawImage(img, 177, 11, 120 , 234, 10, 10, 90, 176);
//tamaño natural
ctx.drawImage(img, 160, 20);
}
```

```
}
```

Este ejemplo dibuja una imagen un par de veces. Primero recorta un área de la imagen original y la escala, por el método de drawImage() que acabamos de relatar. Luego dibuja la imagen original, sin recortar ni escalar, y la coloca al lado de la otra, en el mismo canvas.