TEMA 1 - Parte 4
Contenido multimedia en HTML

1 Inserción de audio y vídeo en HTML5

1.1 ¿Por qué necesitamos estos elementos?

Antes de existir estas etiquetas, si queríamos incluir vídeo en una página web, teníamos que trabajar con un marcado bastante complejo. He aquí un ejemplo tomado directamente de *YouTube*:

```
<object width="425" height="344">
       <param name="movie" value="http://www.youtube.com/v/9sEI1AUFJKw&hl=en_GB&fs=1&">
       </param>
       <param name="allowFullScreen" value="true"></param>
       <param name="allowscriptaccess" value="always"></param>
       <embed src="http://www.youtube.com/v/9sEI1AUFJKw&hl=en_GB&fs=1&"
      type="application/x-shockwave-flash" allowscriptaccess="always"
       allowfullscreen="true" width="425" height="344"></embed>
</object>
```

En primer lugar, tenemos un elemento *<object>*, un contenedor genérico de objetos (que sigue usándose a día de hoy para, por ejemplo, incluir un documento en formato *.pdf* en un sitio web) para incluir la película en Flash. Para evitar inconsistencias entre navegadores, también se incluía un elemento de reserva *<embed>* (que, al igual que *<object>*, también pertenece al estándar) como contenido de repuesto y duplicamos la mayoría de los parámetros del *<object>*.

El código resultante es torpe y no muy legible, y tiene otros problemas: el contenido del *plug-in* no interactúa demasiado bien con las otras tecnologías en la página y tiene problemas de accesibilidad.

1.2 Los elementos *<video>* y *<audio>*

HTML5 introduce **soporte integrado para el contenido multimedia gracias a los elementos *<audio>* y *<video>***, ofreciendo la posibilidad de insertar contenido multimedia en documentos HTML. En primer lugar, a la hora de insertar uno de estos elementos, tanto audio como vídeo, es necesario disponer de él en diferentes formatos para conseguir que se visualice en todos los navegadores. Es posible que haya **discrepancias entre el mismo navegador instalado en diferentes sistemas operativos** ya que para evitar problemas de patentes ciertos **códecs se apoyan** precisamente **en el sistema operativo** y no en el navegador.

Las compatibilidades actuales son las siguientes:

Navegador	Vídeo			Audio		
	MP4	WebM	Ogg	MP3	WAV	Ogg
Microsoft Edge	Sí	No*	No*	Sí	No*	No*
Mozilla Firefox	No*	Sí	Sí	Sí	Sí	Sí
Google Chrome	Sí	Sí	Sí	Sí	Sí	Sí
Safari	Sí	No*	No*	Sí	Sí	No*
Opera	No*	Sí	Sí	Sí	Sí	Sí

**Compatible si se instala manualmente el códec.*

De acuerdo con esto, debemos tener por ejemplo un vídeo llamado ***mi_video*** en al menos dos formatos (lo mismo ocurre con los archivos de audio), pero en nuestro dispondremos de él en tres formatos: ***mi_video.ogg***, ***mi_video.webm*** y ***mi_video.mp4***. Una vez los tenemos, debemos especificar el código para poder reproducirlos correctamente.

Para ello, el código es el siguiente: dentro de la **etiqueta *<video>*** se colocan los **orígenes de datos** utilizando la etiqueta *<source>* correspondientes a los archivos que tengo:

```
<video id="mivideohtml" width="640" height="360" autoplay controls>
       <source src="videos/mi_video.mp4"  type='video/mp4;  codecs="avc1,mp4a"' />
       <source src="videos/mi_video.webm" type='video/webm; codecs="vp9,opus"' />
```

```
      <source src="videos/mi_video.ogg"  type='video/ogg;  codecs="theora,vorbis"' />
      Tu navegador no implementa el elemento <code>video</code>.
</video>
```

Si únicamente tuviera un archivo, podría indicarlo en la propia etiqueta *<video>* o *<audio>* como en el ejemplo siguiente, donde se usa el atributo *src* en vez de usar la etiqueta *<source>*.

```
<audio src="/test/audio.ogg">
      <p>Tu navegador no implementa el elemento <code>audio</code>.</p>
</audio>
```

En el código anterior, usamos `id="mivideohtml"` como el identificador de la etiqueta de vídeo y debe ser único, es decir, **no puede haber dos elementos con el mismo id** ya que, más adelante, cuando vayamos a aplicar estilos y funcionalidades daría problemas.

```
<source src="videos/mi_video.webm" type='video/webm; codecs="vp9,opus"' />
```

El atributo *source* (*src*) es la **ruta en la que está alojada el vídeo**, en este caso es *videos/mi_video.webm*, es decir, en la carpeta */videos* está alojado *mi_video.webm*.

El atributo *type* es donde se define el tipo del archivo, es este caso como el archivo *mi_video.webm* es un archivo de vídeo en formato *.webm*, entonces el valor es *video/webm*. Los diferentes casos son:

- Archivo de vídeo .mp4 → *video/mp4*
- Archivo de vídeo .webm → *video/webm*
- Archivo de vídeo .ogg u .ogv → *video/ogg*
- Archivo de sonido .mp3 → *audio/mp3*
- Archivo de sonido .ogg → *audio/ogg*
- Archivo de sonido .wav → *audio/wav*

Los ***codecs*** son la **codificación que tiene el archivo**, en nuestro caso nuestro el vídeo *mi_video.webm*. Existen muchos tipos de *codecs* pero los más comunes son los que se indican. La manera correcta de definirlos sintácticamente es (con comillas dobles dentro de las simples para seguir la norma de la W3C):

```
type='video/...; codecs="...,..."'
```

Si se define de otra manera (sin poner el signo igual o colocándolo erróneamente) lo que ocurre es que no lee la definición de los *codecs* que se ponen pero aún así lo reproduce, ya que no es indispensable; es como si sólo se indicase:

```
type='video/...'
```

Aunque lo correcto es ponerlo de la primera forma.

```
<video src="video.mp4" width="700" height="350" controls poster="portada.jpg"></video>
<audio src="audio.mp3" preload="auto" loop controls></audio>
```

Otros atributos que pueden usarse (algunos los vemos en los dos ejemplos anteriores) son los siguientes:

- *controls* → muestra los controles estándar de HTML5 para audio o vídeo en una página web.
- *autoplay* → hace que el audio o vídeo se reproduzca automáticamente.
- *loop* → hace que el audio o vídeo se repita automáticamente.
- *poster* → provee una imagen que será mostrada mientras el usuario espera a que cargue el vídeo.

El atributo *preload* es usado en el elemento <*audio*> para almacenar temporalmente (*buffering*) archivos de gran tamaño y puede tomar uno de estos tres valores:

- *none* → no almacena temporalmente el archivo.
- *auto* → almacena temporalmente el archivo multimedia.
- *metadata* → almacena temporalmente sólo los metadatos del archivo.

1.3 Control de la reproducción multimedia

Una vez que se ha incrustado contenido multimedia en el documento, es posible **controlarlo mediante *JavaScript***. Por ejemplo, para iniciar (o reiniciar) la reproducción, puedes hacer uso de:

```
var v = document.getElementsByTagName("video")[0];
v.play();
```

La primera línea obtiene el primer elemento <*video*> en el documento, y la segunda línea llama al método *play()* del elemento, como está definido en la interfaz *HTMLMediaElement* que es usada para implementar los elementos multimedia. Controlar un reproductor de audio, por ejemplo, en HTML5 para **reproducir**, **pausar**, **aumentar** y **disminuir el volumen** usando *JavaScript* es muy sencillo.

```
<audio id="demo" src="audio.mp3"></audio>
<div>
	<button onclick="document.getElementById('demo').play()">
		Reproducir el Audio
	</button>
	<button onclick="document.getElementById('demo').pause()">
		Pausar el Audio
	</button>
	<button onclick="document.getElementById('demo').volume+=0.1">
		Aumentar  Volumen
	</button>
	<button onclick="document.getElementById('demo').volume-=0.1">
		Disminuir  Volumen
	</button>
</div>
```

Mientras que **detener la reproducción multimedia** es tan fácil como llamar al método *pause()* del elemento, el navegador sigue descargando el contenido multimedia hasta que el elemento multimedia es eliminado a través de la recolección de basura. Una forma de detener la descarga de una sola vez sería la siguiente:

```
var mediaElement = document.getElementById("myMediaElementID");
mediaElement.pause();
mediaElement.src = "";
```

Estableciendo una cadena vacía al atributo *src* del elemento multimedia, destruyes el decodificador interno del elemento con lo que detienes la descarga.

Algunos de los **métodos** que se pueden utilizar **para la gestión de medios** con *JavaScript* son los siguientes:

- ***play()***: comienza a reproducir desde el inicio, siempre que el medio no haya sido pausado antes.
- ***pause()***: pausa la reproducción.
- ***load()***: carga el archivo del medio. Es útil para cargar el medio anticipadamente.
- ***canPlayType(formato)***: para comprobar si el formato del archivo se soporta por el navegador.

Por ejemplo, si queremos iniciar la reproducción de un vídeo después de pulsar un botón. Otra opción es usar *.querySelector()*.

```
<video id="video" width="700" height="350">
	<source src="video.mp4">
	<source src="video.ogv">
</video>
<input type="button" id="boton" value="Reproducir">

<script>
	function iniciar() {
		var boton=document.getElementById('boton');
		boton.addEventListener('click', presionar, false);
	}
	function presionar() {
		var video=document.getElementById('video');
		video.play();
	}

	window.addEventListener('load', iniciar, false);
</script>
```

Es posible especificar un **rango de reproducción** agregando los tiempos de inicio y finalización a la URL de la fuente (*src*) con el siguiente formato:

```
#t=[starttime][,endtime]
```

Un ejemplo sería el que aparece a continuación:

```
<source src=http://ejemplo.com/ejemplo1.webm#t=2,3 type=video/webm>
```

2 La etiqueta SVG

SVG es el acrónimo de ***Scalable Vector Graphics***, un estándar del W3C (*World Wide Web Consortium*), **basado en XML**. Utilizamos SVG para **crear imágenes vectoriales bidimensionales** tanto estáticos como animados (estos últimos con la ayuda de **SMIL**). El SVG es un **estándar abierto** y su desarrollo comenzó en 1999. En septiembre de 2001 se convirtió en una recomendación del W3C , por lo que ya había sido incluido de forma nativa en el navegador web del W3C *Amaya*.

Las versiones 1.5 y posteriores de *Mozilla Firefox* soportan gráficos hechos con SVG, así como el navegador Opera que desde su versión 8 ha implementado SVG *1.1 Tiny* en su núcleo. Navegadores como Google Chrome, Safari e Internet Explorer 9 también son capaces de mostrar imágenes en formato SVG sin necesidad de complementos externos. Otros navegadores web, como versiones anteriores a la 9 de Internet Explorer, necesitan un conector o *plug-in*. Hoy en día la mayoría de los navegadores son capaces de reproducir graficos SVG exactamente como si reprodujeran una imagen.

El SVG permite **tres tipos de objetos gráficos**:

- **Elementos geométricos vectoriales** (como caminos consistentes en rectas y curvas, y áreas limitadas por ellos)
- **Imágenes** de mapa de bits/digitales.
- **Texto**.

Los objetos gráficos pueden ser **agrupados**, **transformados** y **compuestos** en objetos previamente renderizados, y pueden recibir un estilo común. El texto puede estar en cualquier espacio de nombres XML admitido por la aplicación, lo que mejora la posibilidad de búsqueda y la accesibilidad de los gráficos SVG.

El juego de características incluye las **transformaciones anidadas**, los *clipping paths*, las máscaras alfa, los filtros de efectos, las plantillas de objetos y la extensibilidad.

2.1 Diferencias entre <*canvas*> y SVG

A diferencia de <*canvas*>, donde los gráficos son dibujados al vuelo (mediante Javascript), en el SVG los elementos pertenecen al DOM (*Document Object Model*), es decir, que si queremos dibujar un círculo, el círculo es en realidad una etiqueta <*circle*>, y es posible referirse al círculo y manipularlo de una manera similar a como se puede manipular un <*div*>.

<*canvas*>	SVG
Dependiente de la resolución.	Independiente de la resolución.
No tiene soporte para manejadores de eventos a nivel de elemento.	Soporte para manejadores de eventos a nivel de elemento.
Pocas opciones para el renderizado de texto.	Adecuado para aplicaciones con grandes áreas que deben renderizarse.
El resultado puede almacenarse como imagen de tipo .png o .jpg.	Si se trata de escenas complejas, el renderizado será muy lento.
Adecuado para juegos con mucha demanda gráfica.	Poco adecuado para aplicaciones de como juegos.

En SVG, cada elemento dibujado se recuerda como un objeto. Si un atributo del objeto SVG se cambia, el **navegador automáticamente renderiza** (redibuja) de nuevo **el elemento**. En <*canvas*>, en cambio, se renderiza **la escena píxel a píxel**. Una vez que se ha dibujado el objeto, el navegador lo olvida. Si algo cambia, toda la escena se redibuja.

2.2 Creación de una imagen con SVG

2.2.1 El sistema de coordenadas en SVG

El origen del sistema de coordenadas en SVG es similar al de <*canvas*>, siendo la esquina superior izquierda del lienzo. Por lo tanto, **el eje y aparece al revés comparado con un sistema normal de coordenadas**.

2.2.2 Creación de documentos en SVG

Podemos fácilmente crear documentos SVG utilizando esta estructura:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:ev="http://www.w3.org/2001/xml-events" baseProfile="full">
       <!-- el contenido del docunento SVG-->
</svg>
```

El documento tiene que llevar la **extensión .svg**. El documento SVG puede ser abierto directamente en el navegador. Además, puede ser **utilizado como imagen en un documento HTML ya existente**.

```
<img src="prueba.svg" alt="ejemplo" height="200" width="250">
```

Incluso podemos utilizar el elemento <*iframe*> con este propósito.

```
<iframe src="test.svg" width="250" height="200" >
```

Otra manera de utilizar las imágenes SVG es convertirlas a **data:uri** (*Uniform Resource Identifier*).

1. Una opción es utilizar código **SVG no codificado** (*unencoded*) así:

```
.data-uri-uncoded {
   background-image: url('data:image/svg+xml; utf8,
   <svg xmlns = "http://www.w3.org/2000/svg"> . . . </svg>');
}
<div class="data-uri-uncoded"></div>
```

2. Otra propuesta sería convertir primero el código **SVG a base64** y después utilizarlo.

```
<a href=". . . ">Imagen SVG</a>
```

En Internet encontramos herramientas que **codifican una cadena de texto en base64**:

https://www.base64encode.org/
http://www.mobilefish.com/services/base64/base64.php

También podemos utilizar **PHP** con este propósito utilizando la función *base64_encode*.

```
$svg img ="<svg xmlns='http://www.w3.org/2000/svg'> ... </svg>";
$data_URI = base64_encode($svg_img);
echo "<a href='".$data_URI."'>Mira la URL</a>";
```

2.2.3 SVG en HTML5

HTML5 tiene soporte para SVG incrustado (*inline*). En el HTML5, podemos incrustar elementos SVG directamente en las páginas HTML.

```
<svg width="250" height="100">
      <circle cx="125" cy="50" r="40" stroke="#030" stroke-width="4" fill="#6AB150" />
</svg>
```

La anchura y la altura del elemento pueden ser **definidos con los atributos *width* y *height*** como en el ejemplo anterior, pero también podemos utilizar el CSS con este propósito.

```
<svg style="width:250; height:100;">
      <circle cx="125" cy="50" r="40" stroke="#030" stroke-width="4" fill="#6AB150">
      </circle>
</svg>
```

- El **elemento *<circle>*** se utiliza para **dibujar un círculo**.
- Los **atributos *cx* y *cy* representan las coordenadas *x* e *y* del centro del círculo**. Si *cx* y *cy* se omiten, el centro del círculo está en el punto *x = 0, y = 0*.
- El **atributo *r*** define el **radio del círculo** (en este caso, 40px).
- El **atributo *stroke*** define el **color del trazado**.
- El **atributo *stroke-width*** define el **grosor del trazado** (en este caso, 4px).
- El **atributo *fill*** define el *color de relleno* (en este caso #6AB150).

En caso de tener más de un elemento, aquellos definidos después se superponen sobre los primeros.

```
<svg height="150" width="500">
      <ellipse cx="240" cy="100" rx="220" ry="30" style="fill:purple" />
      <ellipse cx="220" cy="70" rx="190" ry="20" style="fill:lime" />
      <ellipse cx="210" cy="45" rx="170" ry="15" style="fill:yellow" />
</svg>
```

2.3 Ejemplos de figuras mediante SVG

2.3.1 Rectángulo

```
<svg width="400" height="100">
       <rect width="400" height="100" style="fill:rgb(0,0,255);
       stroke-width:10; stroke:rgb(0,0,0)" />
</svg>
```

2.3.2 Rectángulo con esquinas redondeadas

```
<svg width="400" height="180">
       <rect x="50" y="20" rx="20" ry="20" width="150" height="150" style="fill:red;
       stroke:black; stroke-width:5; opacity:0.5" />
</svg>
```

Los atributos *rx* y *ry* indican los radios para las esquinas del rectángulo, siendo *rx* el radio correspondiente al eje *x* y *ry* el correspondiente al eje *y*, de forma similar a como ocurre en una elipse. Si *ry* no se define, se toma que *rx = ry*.

2.3.3 Elipse (SVG Logo)

```
<svg height="130" width="500">
       <defs>
              <linearGradient id="grad1" x1="0 %" y1="0 %" x2="100 %" y2="0 %">
                     <stop offset="0%" style="stop-color:rgb(255,255,0); stop-
                     opacity:1" />
                     <stop offset="100%" style="stop-color:rgb(255,0,0); stop-
                     opacity:1" />
              </linearGradient>
       </defs>
       <ellipse cx="100" cy="70" rx="85" ry="55" fill="url(#grad1)" />
       <text fill="#ffffff" font-size="45" font-family="Verdana" x="50" y="86">
              SVG
       </text>
</svg>
```

Los atributos *rx* y *ry* indican los radios en los ejes x e y para la elipse.

2.3.4 Polígono (estrella)

```
<svg width="300" height="200">
       <polygon points="100,10 40,198 190,78 10,78 160,198"
       style="fill:lime; stroke:purple; stroke-width:5; fill-rule:evenodd;" />
</svg>
```

El **atributo *fill-rule*** indica el algoritmo que se usa para determinar **qué lado de la ruta está dentro de la forma**. Para una forma simple, intuitivamente se ve claro cuál es la región que permanece en la parte interior de la figura, pero sin embargo, para rutas más complejas, especialmente si una línea de la ruta intersecta consigo misma o si una ruta tiene en su interior varias rutas (subrutas), la interpretación de dentro o fuera no es tan obvia. Los valores que puede tomar son ***evenodd*** y ***nonzero***, o permitir que herede del elemento padre (***inherit***).

2.3.5 Polilínea

La única diferencia entre *<polygon>* y *<polyline>* es que en *<polygon>* se cierra el trazado, mientras que *<polyline>* no.

```
<svg width="250" height="150">
       <polyline points="125,10 175,130 75,130"
       style="fill:#6AB150; stroke:#003300; stroke-width:3" />
</svg>
```

2.4 Creación de gradientes

Además de colores sólidos, en SVG podemos **crear degradados lineales y radiales**. Para esto utilizamos los siguientes **elementos *<linearGradient>*** para crear gradientes lineales **y *<radialGradient>*** para generar gradientes radiales.

El **elemento *<stop>*** se utiliza para **especificar los colores o las transparencias** utilizadas para el gradiente y dónde éstos acaban.

```
<svg width="250" height="100">
       <defs>
              <linearGradient id="linearGradient">
                     <stop offset="10%" stop-color="red"></stop>
                     <stop offset="30%" stop-color="blue"></stop>
                     <stop offset="50%" stop-color="green"></stop>
                     <stop offset="70%" stop-color="yellow"></stop>
                     <stop offset="90%" stop-color="red"></stop>
              </linearGradient>
       </defs>
       <rect fill="url(#linearGradient)" width="250" height="100"></rect>
</svg>
```

En el elemento *<linearGradient>* podemos utilizar el atributo *gradientTransform="rotate(30 .5 .5)"* que permite girar 30º en el sentido de las agujas del reloj el degradado.

El **atributo *stop-opacity***, también en un **degradado**, controla la **opacidad/transparencia del color de relleno** y puede tomar valores entre 0 (transparente) y 1 (opaco) y crea gradientes (lineales o radiales) de transparencias.

```
<svg width="250" height="150">
       <defs>
              <radialGradient id="radialGradient5">
                     <stop offset="0%"   stop-color="chartreuse"></stop>
                     <stop offset="100%" stop-color="#030"></stop>
              </radialGradient>
       </defs>
       <rect fill="url(#radialGradient5)" width="250" height="150"></rect>
</svg>
```

Los **atributos *fx* y *fy*** se utilizan junto con el elemento *<radialGradient>* para definir el **foco del gradiente radial**. El **valor de los atributos *fx* y *fy* es un número de 0 a 1** y, si no están definidos, el foco coincide por defecto con el centro de la elipse.

```
<radialGradient id="radialGradient7" fx=".95">
```

2.5 Patrones de relleno

En el siguiente ejemplo utilizamos una imagen como patrón de relleno. La imagen definida con *<image>* aparece dentro del elemento *<pattern>* que a su vez aparece definido dentro del elemento *<defs>*. Al elemento *<pattern>* le asignamos el *id="imagen"* para poder referenciarlo más tarde.

Además, en el lienzo dibujamos un **rectángulo** que utiliza como relleno el patrón definido anteriormente.

```
<svg width="250" height="200">
       <defs>
```

```
            <pattern id="imagen0" width="125" height="131"
            patternUnits="userSpaceOnUse">
                  <image xlink:href="images/ejemplo1.jpg" width="125" height="131">
                  </image>
            </pattern>
      </defs>
      <rect width="250" height="200" fill="url(#imagen0)">
      </rect>
</svg>
```

También es posible usar otros elementos como patrones de relleno. En este caso una **elipse**:

```
<svg width="250" height="120">
      <defs>
            <pattern id="elipse2" x="5" y="0" width="40" height="20"
             patternUnits="userSpaceOnUse">
                  <ellipse cx="20" cy="10" rx="20" ry="10" fill="#6ab150">
                  </ellipse>
            </pattern>
      </defs>
      <rect x="25" y="10" width="200" height="100" fill="url(#elipse2)" stroke="#f00">
      </rect>
</svg>
```

3 Introducción al elemento <*canvas*>

El elemento <*canvas*> de HTML5 proporciona una manera fácil y potente de dibujar gráficos usando *JavaScript*. Cada elemento <*canvas*> utiliza un contexto (sería como una página de un cuaderno de dibujo), en el que podemos lanzar comandos de *JavaScript* para dibujar lo que queramos. Los navegadores pueden aplicar múltiples contextos <*canvas*> y las diferentes APIs proporcionan la funcionalidad de dibujo.

El elemento <*canvas*> puede definirse como un **entorno para crear imágenes dinámicas**. El propio elemento es relativamente simple. Lo **único que hay que especificar** al usarlo son sus **dimensiones**:

```
<canvas id="entorno_canvas" width="360" height="240">
</canvas>
```

Cualquier cosa que escribamos entre la apertura y cierre de la etiqueta <*canvas*> solamente será interpretada por navegadores que no soportan aún la nueva etiqueta.

```
<canvas id="entorno_canvas" width="360" height="240">
      <p>Tu navegador no soporta canvas: deberías de ver una imagen.</p>
      <img src="html5_logo.png" alt="HTML5" />
</canvas>
```

En el ejemplo, la imagen solo sería visible por aquellos navegadores que no soporten <*canvas*>.

3.1 ¿Cómo se usa?

Cuando trabajamos con <*canvas*>, realmente usamos Javascript. Primero debemos **referenciar el elemento y adquirir su contexto**. Por el momento, el único contexto disponible es el **contexto bidimensional**, aunque en la especificación de HTML se habla de una futura definición de un contexto 3D.

```
var canvas = document.getElementById("entorno_canvas");
var ctx = canvas.getContext("2d");
```

Una vez adquirimos el contexto, podemos empezar a dibujar en la superficie del <*canvas*>. La API bidimensional ofrece muchas de las **herramientas que podemos encontrar en cualquier aplicación de diseño gráfico**: trazos, rellenos, gradientes, sombras, formas y curvas Bézier.

Una vez definido el <*canvas*> y su contexto, es en éste último donde **se definen los métodos y propiedades para dibujar**. Por ejemplo tenemos algunas dedicadas a los rectángulos:

- La propiedad ***fillStyle*** puede ser un **color** (definido como en CSS), un **patrón** o un **gradiente**. Por defecto es negro, y cada contexto recuerda sus propiedades mientras la página donde se definió el <*canvas*> esté abierta o no se haya reseteado éste de algún modo. Es posible usar **rgb**, **rgba**, un código **hexadecimal** o la **palabra clave** de un color.
- ***fillRect (x, y, ancho, alto)*** dibuja un **rectángulo relleno** con el estilo de relleno actual.
- La propiedad ***strokeStyle*** es similar a ***fillStyle***. Puede ser un **color**, un **patrón** o un **gradiente**.
- ***strokeRect (x, y, ancho, alto)*** muestra un rectángulo con el estilo actual de trazo, pero únicamente sus **bordes**.
- Mediante ***lineWidth*** especificamos el **ancho de su contorno**.
- ***clearRect (x, y, ancho, alto)*** **limpia los píxeles del rectángulo** especificado

El <*canvas*> internamente es una r**ejilla bidimensional y la esquina superior izquierda tiene las coordenadas (0,0)**, que se van incrementando hacia la derecha y hacia abajo.

```
ctx.fillStyle = "rgba(0, 0, 0, 0.5)";
ctx.fillRect(100, 10, 60, 50);
ctx.lineWidth = 5
ctx.strokeStyle = "red";
ctx.strokeRect(200, 70, 60, 50);
ctx.clearRect(210, 20, 25, 25);
```

3.2 Dibujar con *JavaScript*

En primer lugar vamos a especificar el color del trazo:

```
context.strokeStyle = '#8f9814';
```

Cualquier cosa que dibujemos tendrá un contorno de color verde. Para visualizar este ejemplo, vamos a dibujar un rectángulo:

```
context.strokeRect(20, 30, 100, 50);
```

La sintaxis del método *strokeRect* es como sigue: ***strokeRect (izquierda, arriba, ancho, largo)***. Por lo tanto, en el ejemplo anterior, hemos dibujado un rectángulo posicionado a 20 píxeles del margen izquierdo y 30 píxeles del derecho con un tamaño de 100 × 50 píxeles.

Si queremos dibujar un rectángulo relleno con color, deberemos de especificar el color del relleno:

```
context.fillStyle = '#FF8F43';
context.fillRect(20, 30, 100, 50);
```

Y si queremos dibujar un rectángulo con trazo y relleno debemos generar uno dentro del otro:

```
context.fillRect(20, 30, 100, 50);
context.strokeRect(19, 29, 101, 51);
```

3.3 Funcionamiento de <*canvas*>

El potencial de <*canvas*> reside en su habilidad para **actualizar su contenido en tiempo real**. Si la usamos para responder a eventos de usuario, podemos crear herramientas y juegos que hubiesen requerido de un *plug-in* externo como *Flash*.

```
<canvas height="100" id="myCanvas" width="200">
	Tu navegador no soporta la etiqueta canvas de HTML5.
</canvas>
<script type="text/javascript">
	var c = document.getElementById("myCanvas");
	var cxt = c.getContext("2d");
	cxt.fillStyle = "#FF0000";
	cxt.fillRect(0, 0, 150, 75);
</script>
```

3.3.1 Líneas

Para indicar que queremos dibujar una línea, lo primero que debemos hacer es llamar a la función ***beginPath()*** y, a continuación, llamaremos a ***moveTo(x, y)*** para movernos a la posición donde nuestra línea comenzará. Seguidamente, utilizaremos la función ***lineTo(x, y)*** para dibujar la línea como tal y la haremos visible con ***stroke()***. Podemos especificar el ancho y el color de la línea con las propiedades *lineWidth* y *strokeStyle*.

```
ctx.beginPath();
ctx.moveTo(5, 5);
ctx.lineTo(5, 295);
ctx.moveTo(295, 5);
ctx.lineTo(295, 295);
ctx.lineWidth = 1;
ctx.strokeStyle = "#CC0000";
ctx.stroke();
```

Ahora, creamos un triángulo desde un punto especifico del *<canvas>*. Después, con *lineTo()* dibujamos las líneas que forman el triángulo y, finalmente, con *closePath*() creamos un camino que regresa hasta el punto de inicio.

```
ctx.beginPath();
ctx.moveTo(300, 200);
ctx.lineTo(200, 150);
ctx.lineTo(200, 0);
ctx.closePath();
ctx.stroke();
```

3.3.2 Arcos

Para indicar que vamos a dibujar un arco llamaremos inicialmente a la función *beginPath()*. Dibujaremos un arco o un círculo con la función ***arc(x, y, radio, angulo-inicio, angulo-fin, sentido)***. Las coordenadas *x* e *y* indican el **centro del círculo**, y los **ángulos han de indicarse en radianes**. El arco no será visible hasta llamar a ***fill()*** o a ***stroke()***.

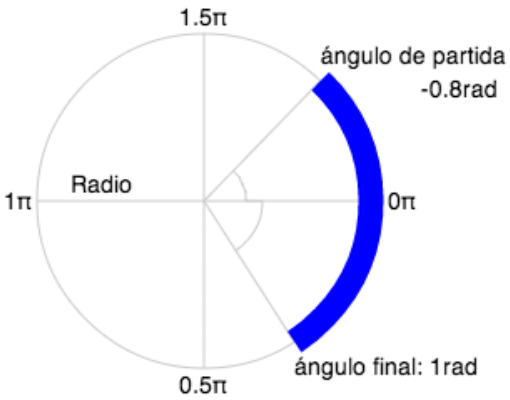
El **parámetro *sentido*** puede tomar dos valores: *true* (verdadero, es decir, en **sentido contrario al de las agujas del reloj**) y *false* (falso, **en el sentido de las agujas del reloj**).

```
ctx.beginPath();
ctx.arc(150, 150, 50, 0, 2 * Math.PI);
ctx.lineWith = 10;
ctx.strokeStyle = "yellow";
ctx.stroke();

ctx.fillStyle = "#FF0000";
ctx.beginPath();
ctx.arc(70, 18, 15, 0, Math.PI * 2, true);
ctx.closePath();
ctx.fill();
```

En <*canvas*> para trabajar con ángulos, utilizamos radianes. Un círculo completo tiene 2π radianes, con lo que podemos escribir ***360° = 2π rad*** o, en JavaScript, ***2 * Math.PI***. Para convertir grados sexagesimales a radianes utilizamos la siguiente fórmula: *radianes = (π / 180) * grados*; lo que traducido a JavaScript es:

```
var radianes = (Math.PI / 180) * grados;
```




```
for (var i = 0; i < 4; i++) {
for(var j = 0; j < 3; j++) {
              ctx.beginPath();
              var x = 25 + j * 50;
              var y = 25 + i * 50;
              var radius         = 20;
              var start_angle    = 0;
              var end_angle      = Math.PI + (Math.PI * j) / 2;
              var anticlockwise  = i % 2 == 1;
              ctx.arc(x, y, radius, start_angle, end_angle, anticlockwise);
              if (i > 1) {
                     ctx.fill();
              } else {
                     ctx.stroke();
              }
       }
}
```

3.3.3 Texto

Mediante las funciones ***fillText(texto, x, y)*** y ***strokeText(texto, x, y)*** podemos **dibujar texto** dentro del <*canvas*>. Con la primera de ellas damos forma al texto y lo rellenamos con el valor de *fillStyle* actual, mientras que con la segunda dibujamos únicamente el perfil del texto con el valor de *strokeStyle* actual. Existe un cuarto parámetro opcional que indica la anchura máxima, que hace que el navegador reduzca el tamaño del texto para que quepa en el ancho dado si fuera necesario. Podemos dar formato al texto estableciendo previamente las propiedades *font*, *textAlign*, *textBaseline.*

```
ctx.font = 'Bold 20px Tahoma';
ctx.textAlign = 'center';
ctx.fillStyle = 'black';
ctx.fillText('INTERFAZ', 50, 50);

var text = "Ejemplo";
ctx.font = "20pt Verdana";
ctx.textAlign = "center";
ctx.fillStyle = "blue";
ctx.fillText(text, canvas.width / 2, (canvas.height / 2) - 2);
ctx.translate(canvas.width / 2, (canvas.height / 2) + 2);
ctx.scale(1, -1);
```

```
ctx.globalAlpha = 0.1;
ctx.fillText(text,0,0);
```

3.3.4 Imágenes

Para añadir una imagen lo primero que tenemos que debemos hacer es **crear un objeto *Image*** y establecer su propiedad *src*. Con la función *drawImage(imagen, x, y)* añadiremos la imagen al <*canvas*>.

```
var img = new Image();
img.src = 'img/ejemplo1.png';
img.onload = function(){
       ctx.drawImage(img, 10, 10);
};
```

Si ejecutamos el método *.drawImage()* sin llamar al evento *.onload* puede que no se ejecute correctamente si todavía no se ha cargado la imagen.

3.4 Uso de gradientes, sombras y transparencia

Los gradientes lineales son definidos por una línea imaginaria que nos da la dirección del gradiente. El método *createLinearGradient(x0, y0, x1, y1)* toma cuatro argumentos. Los dos primeros (*x0,y0*) representan las **coordenadas del origen de nuestra línea**, mientras que los dos siguientes dos (*x1,y1*) representan las coordenadas del punto donde esta línea acaba.

El método *addColorStop(stop, color)* toma dos argumentos. El primer argumento, *stop*, puede tomar **valores desde 0 hasta 1**, y representa una **posición a lo largo del vector gradiente** definido anteriormente con método *createLinearGradient()*. Podemos utilizar el método *addColorStop()* tantas veces como creamos necesario.

El segundo argumento representa el **color**, que puede tomar **cualquier valor aceptado en el CSS**.

```
var grd = ctx.createLinearGradient(0, 0, 175, 50);
grd.addColorStop(0, "#FF0000");
grd.addColorStop(1, "#00FF00");
ctx.fillStyle = grd;
ctx.fillRect(0, 0, 175, 50);

ctx.rect(0, 0, canvas.width, canvas.height);
var grd = ctx.createRadialGradient(80, 125, 25, 125, 125, 120);
grd.addColorStop(0, "#8DF");
grd.addColorStop(1, "#04B");
ctx.fillStyle = grd;
ctx.fill();
```

Otra posibilidad que tenemos es la de añadir sombra a un objeto. Para ellos utilizaremos las funciones *shadowColor* y *shadowBlur*. La primera permite **seleccionar el color de la sombra**; la segunda, **ajustar el grado de desenfoque**. Mientras, con el método *fillRect(x, y, ancho, alto)*, damos una posición y dimensiones a la figura. Además, tenemos también las propiedades *shadowOffsetX* y *shadowOffsetY* que especifican el desplazamiento *x* e *y* de la sombra en píxeles.

```
ctx.shadowBlur = 30;
ctx.shadowColor = "black";
ctx.fillStyle = "blue";
ctx.fillRect(50, 50, 350, 200);
```

Además, por supuesto, podemos utilizar distintas sentencias de *JavaScript*, como bucles. Para complicarlo un poco, podemos realizar el camino con un bucle (un bucle *for* en este caso), en el que en cada iteración dibujaremos un segmento del camino (mediante una instrucción *if*). El resultado es un perfil de una escalera.

```
ctx.beginPath();
ctx.moveTo(1, 1);
for (i = 1; i < 100; i += 5) {
	if((i % 2) != 0) {
		ctx.lineTo(i + 5, i);
}
	else {
		ctx.lineTo(i, i + 5);
	}
}
ctx.lineTo(1, 100);
ctx.closePath();
ctx.stroke();
```

3.4.1 Transparencia (*globalAlpha*)

La propiedad *globalAlpha* determina o devuelve el valor *alfa* o **la opacidad real del dibujo**. Puede tomar valores entre 0 (totalmente transparente) y 1 (totalmente opaco).

```
<canvas width="250" height="150" id="lienzo1">No tiene soporte de canvas</canvas>
var canvas = document.getElementById("lienzo1");
if (canvas && canvas.getContext) {
	var ctx = canvas.getContext("2d");
	if (ctx) {
		ctx.globalAlpha = 0.75;
		ctx.fillStyle   = "#F00";
		ctx.fillRect(60,20,75,50);
		ctx.fillStyle   = "#0F0";
		ctx.fillRect(90,50,75,50);
		ctx.fillStyle   = "#00F";
		ctx.fillRect(120,80,75,50);
	}
}
```

3.5 Creación de patrones

Un patrón puede **ser creado a partir de una imagen, un video, o incluso de otro *<canvas>***, y puede ser configurado para repetirse en todas las direcciones (*repeat*), repetirse en una sola dirección, (en *repeat-x* o en *repeat-y*). Incluso puede ser configurado para que no se repita (*no-repeat*).

Para crear patrones utilizamos el método ***createPattern(img, dir)***. El primer argumento representa la **imagen** que utilizamos para nuestro patrón. El segundo representa la **dirección**, y puede tomar los siguientes valores, que tienen el mismo significado que en CSS: *no-repeat*, *repeat*, *repeat-x* y *repeat-y*.

3.5.1 Crear un patrón desde una imagen

Cuando creamos un patrón desde una imagen es mejor hacerlo **cuando la imagen se haya cargado** (*img.onload*), ya que las imágenes pueden cargarse lentamente, y probar a crear un patrón desde una imagen inexistente, resultaría en error. Para que esto no suceda, tenemos que escribir una función anónima que crea el patrón, y enlazamos esta función como un *callback* al evento *onload*.

Creamos el patrón (*pattern*) utilizando el método *createPattern()* y asignamos su valor a la propiedad *fillStyle()*, para un relleno, o a la propiedad *strokeStyle()* para dibujar un borde.

```
var canvas = document.getElementById("lienzo");
if (canvas && canvas.getContext) {
	var ctx = canvas.getContext("2d");
	if (ctx) {
		var patron = new Image();
```

```
		patron.src = "images/pattern.png";
		patron.onload = function() {
			ctx.fillStyle = ctx.createPattern(patron,"repeat");
			ctx.fillRect(50, 50, canvas.width-100, canvas.height-100);
			ctx.strokeStyle = ctx.createPattern(patron,"repeat");
			ctx.lineWidth = 20;
			ctx.strokeRect(10, 10, canvas.width-20, canvas.height-20);
		}
	}
}
```

3.5.2 Crear un patrón desde otro <*canvas*>

También podemos **crear patrones utilizando otro <*canvas*> en vez de una imagen**. Primero dibujamos en un pequeño <*canvas*> (*id="patron" width="10" height="15"*) una línea oblicua. Ésta será la imagen que se repetirá para crear un patrón en el otro canvas (*id="lienzo1" width="200" height="160"*). Éste podría ser un <*canvas*> virtual, creado con ***document.createElement("canvas")*** y que no aparece inicialmente en el DOM.

```
<canvas id="patron" width="10" height="15">No tiene soporte para canvas</canvas>
<canvas id="lienzo" width="200" height="160">No tiene soporte para canvas</canvas>
var canvasP = document.getElementById("patron");
if (canvasP && canvasP.getContext) {
	var ctxP = canvasP.getContext("2d");
	if (ctxP) {
		ctxP.lineWidth = 1;
		ctxP.strokeStyle = "#ccc";
		ctxP.beginPath();
		ctxP.moveTo(canvasP.width, 0);
		ctxP.lineTo(0, canvasP.height);
		ctxP.stroke();
	}
}

var canvas1 = document.getElementById("lienzo1");
if (canvas1 && canvas1.getContext) {
	var ctx1 = canvas1.getContext("2d");
	if (ctx1) {
		ctx1.save();
		ctx1.fillStyle = ctx1.createPattern(canvasP,"repeat");
		ctx1.fillRect(0,0, canvas1.width, canvas1.height);
		ctx1.restore();
		ctx1.font = "18px Arial";
		ctx1.textAlign = "center";
		ctx1.textBaseline = "middle";
		ctx1.fillText("lienzo1", canvas1.width/2, canvas1.height/2);
	}
}
```

3.6 Manipulación de imágenes

También podemos utilizar imágenes en el <*canvas*>. Lo podemos hacer utilizando las reglas del CSS para establecer una imagen como fondo del <*canvas*>, o lo podemos hacer de manera dinámica utilizando métodos y propiedades especificas. En el primer caso se haría de la siguiente forma:

```
canvas { background-image: url(img.png); }
```

En este segundo supuesto, podemos utilizar:

- Una imagen desde archivo.
- Una imagen ya existente en la página.
- Una imagen dibujada en otro <*canvas*>.

También podemos redimensionar, recortar o reposicionar esta imagen.

Cuando queremos utilizar una imagen desde un archivo, hay que llamar **el método *drawImage()*** solo cuando la imagen se haya cargado (*img.onload*), ya que las imágenes pueden cargarse lentamente, y probar a utilizar una imagen inexistente generaría un error. Para que esto no suceda, debemos escribir una función anónima que utiliza *drawImage()*, y enlazar esta función como *callback* al evento *onload*.

Para ello, utilizamos el método *drawImage(img,x,y)* en su forma más sencilla: **con tres argumentos**.

```
<canvas id="lienzo" width="250" height="232">No tiene soporte para canvas</canvas>
window.onload = function() {
var canvas = document.getElementById("lienzo");
	if (canvas && canvas.getContext) {
		var ctx = canvas.getContext("2d");
		if (ctx) {
			var img=new Image();
			img.src = "images/enfeinada250.jpg";
			img.onload = function() {
				ctx.drawImage(this, 0, 0);
			}
		}
	}
}
```

Se trata de un método bastante complejo que, como ya hemos visto puede tomar tres argumentos, pero también cinco o nueve argumentos:

- **Tres argumentos**: el **uso básico de *drawImage()*** consiste en un argumento para apuntar a la imagen a incluir y dos para especificar las coordenadas de destino dentro del contexto <*canvas*>.
- **Cinco argumentos**: el **uso medio de *drawImage()*** incluye esos tres argumentos, más dos para especificar la **anchura y la altura de la imagen insertada** (en caso de que se desee cambiar su tamaño).
- **Nueve argumentos**: el **uso más avanzado de *drawImage()*** incluye los anteriores cinco argumentos, más **dos valores de coordenadas dentro de las imágenes de origen** y **dos valores para la anchura y la altura dentro de la imagen de origen**. Estos valores permiten recortar de forma dinámica la imagen de origen antes de ponerla en el contexto <*canvas*>.

El siguiente código de ejemplo muestra los tres tipos de *drawImage()* en acción:

```
context.drawImage(img_elem, dx, dy);
context.drawImage(img_elem, dx, dy, dw, dh);
context.drawImage(img_elem, sx, sy, sw, sh, dx, dy, dw, dh);
```

3.7 Manipulación basada en píxeles

La API del contexto 2D ofrece tres métodos para dibujar píxel a píxel: ***createImageData()***, ***getImageData()*** y ***putImageData()***. Los píxeles en bruto se guardan en objetos de tipo ***ImageData***. Cada objeto tiene tres propiedades: *width*, *height* y *data*. La propiedad *data* es de tipo *CanvasPixelArray*, y almacena una cantidad de elementos igual a *width* × *height* × 4 (4 bytes, como ahora veremos), lo que significa que para cada píxel se definen los valores de rojo, verde, azul y alfa, en el orden en que queramos que aparezcan (todos los valores van de 0 a 255, incluyendo alfa). Los píxeles se ordenan de izquierda a derecha, fila por fila, de arriba a abajo. Vamos a ver un ejemplo paso a paso para comprenderlo mejor.

El elemento <*canvas*> **nos permite modificar, uno por uno, los píxeles de una imagen**. Para esto tenemos a nuestra disposición **tres propiedades y tres métodos** de <*canvas*>. Los **píxeles de una imagen** tienen una anchura de **4 bytes**, uno por cada componente **R G B A**, *Red* (rojo), *Green* (verde), *Blue* (azul) y *Alpha* (transparencia). Accediendo uno por uno los píxeles de una imagen podemos modificar estos componentes de color, y por tanto manipular el aspecto de las imágenes en el <*canvas*>.

A continuación, dibujamos un cuadrado utilizando el método ***createImageData()***. Sería más fácil dibujarlo con *fillRect()*, pero es para demonstrar que podemos dibujar una imagen sencilla partiendo prácticamente desde cero modificando cada uno de sus píxeles.

1. El método *createImageData(anchura , altura)* crea un nuevo objeto *ImageData* en blanco, de anchura y altura dadas.
2. Iteramos de píxel en píxel modificando uno por uno el valor de los componentes RGBA.

   Cada píxel una anchura de 4 bytes, donde el **primer byte representa el rojo**, el segundo representa el **verde**, el tercero el **azul** y el cuarto es el componente ***alpha*** o el grado de transparencia. Siendo bytes pueden tomar **valores entre 0 y 255**.

   En el siguiente ejemplo manipulamos cada pixel para que:

   1. **Rojo** = 255 → 100%.
   2. **Verde** = 0 → ausente.
   3. **Azul** = 0 → ausente.
   4. **Alpha** = 255 → totalmente opaco .
3. Después de manipular los pixeles volvemos a colocar la imagen en el <*canvas*> con *putImageData()*.

El método *putImageData()* utiliza (en este caso) tres argumentos: la imagen manipulada y las coordenadas *x* e *y* de donde poner la imagen en el <*canvas*>.

```
<canvas width="250" height="120" id="lienzo">No tiene soporte para canvas</canvas>

var canvas = document.getElementById("lienzo");
if (canvas && canvas.getContext) {
	var ctx = canvas.getContext("2d");
	if (ctx) {
		var imgData=ctx.createImageData(100,100);
		for (var i = 0; i < imgData.data.length; i+= 4)
		{
			imgData.data[i+0]=255;	// rojo = 100%;
			imgData.data[i+1]=0;	// verde - ausente;
			imgData.data[i+2]=0;	// azul - ausente;
			imgData.data[i+3]=255;	// alpha - opaco;
		}
		ctx.putImageData(imgData,75,10);
	}
}
```

3.7.1 Las propiedades *ImageData.width* e *ImageData.height*

En el ejemplo anterior, hemos utilizado la propiedad *data* del objeto ***ImageData*** que representa un ***array* unidimensional** que contiene los **datos rgba de cada píxel**. La longitud de este *array* es igual al número de pixeles multiplicado por 4.

Otra manera de hacer lo mismo es utilizando las propiedades ***width*** (anchura) y ***height*** (altura) del objeto *ImageData*.

Si en el ejemplo anterior hemos utilizado un solo bucle *for*, ahora necesitamos utilizar dos bucles *for* anidados: uno para las coordenadas *x* y otro para las coordenadas *y* de cada píxel.

```
for (var y = 0; y < imgData.height; y ++) {          //imgData.height = 250
       for (var x = 0; x < imgData.width; x ++) {    //imgData.width = 250
              // aquí va el código
       }
}
```

La fórmula para encontrar el índice de los datos en el *array ImageData.data* es:

```
var i = (y * imgData.width + x) * 4;
```

El resto es similar al ejemplo anterior:

```
for (var y = 0; y < imgData.height; y ++) {
       for (var x = 0; x < imgData.width; x ++) {
              var i = (y * imgData.width + x) * 4;
              imgData.data[i + 0] = 255;       // rojo = 100%;
              imgData.data[i + 1] = 0;         // verde - ausente;
              imgData.data[i + 2] = 0;         // azul - ausente;
              imgData.data[i + 3] = 255;       // alpha - opaco;
       }
}
ctx.putImageData(imgData,75,75);
```

3.7.2 Convertir una imagen a blanco y negro

Para convertir a blanco y negro una imagen en colores, hay que **manipular uno por uno cada píxel** de esta imagen y cambiar el color del píxel por el valor calculado de la luminosidad (*brightness*) del mismo. Podemos considerar la **luminosidad** como un **promedio de los valores de cada color**.

```
var luminosidad = parseInt((pixels[i] + pixels[i + 1] + pixels[i + 2]) / 3);
```

Aunque también es posible utilizar otras fórmulas.

```
<img id="alFaro" src="images/al_faro.jpg" alt="al faro">
<canvas width="250" height="188" id="lienzo">No tiene soporte para canvas</canvas>

window.onload = function()
{
       var canvas = document.getElementById("lienzo");
       if (canvas && canvas.getContext)
       {
              var ctx = canvas.getContext("2d");
              var srcImg = document.getElementById("alFaro");
              ctx.drawImage(srcImg, 0, 0, ctx.canvas.width, ctx.canvas.height);

              var imgData = ctx.getImageData(0, 0, ctx.canvas.width, ctx.canvas.height);
              var pixels = imgData.data;

              for (var i = 0; i < pixels.length; i += 4) {
                     var luminosidad = parseInt((pixels[i] + pixels[i+1] +
                     pixels[i+2])/3);
                     pixels[i]    = luminosidad;      // rojo
                     pixels[i + 1]= luminosidad;      // verde
                     pixels[i + 2]= luminosidad;      // azul
              }
              ctx.putImageData(imgData, 0, 0);
       }
}
```