


Por *Jesús Gabriel Rivera*

En  Github como `grChad`

Me he basado en información proporcionada por MDN (Mozilla Developer Network).

Tutorial de Canvas



`<canvas>` es un elemento **HTML** el cual puede ser usado para dibujar gráficos usando `scripts` (normalmente **JavaScript**). Este puede, por ejemplo, ser usado para dibujar gráficos, realizar composición de fotos simples (y no tan simples) animaciones.

En este tutorial se describe cómo usar el elemento `<canvas>` para dibujar gráficos en 2D, empezando con lo básico. Los ejemplos le proveerán mayor claridad a las ideas que pueda tener referentes al `canvas`, así como los códigos que necesita para crear su propio contenido.

1. Uso básico de Canvas

Comenzamos observando el elemento `<canvas>`. Al final de esta sección, sabrás como configurar el entorno 2D de `canvas` y habrás dibujado el primer ejemplo en tu navegador.

1.1. El elemento canvas

```
<canvas id="tutorial" width="150" height="150"></canvas>
```

A primera vista, un elemento `<canvas>` es parecido al elemento ``, con la diferencia que este no tiene los atributos `src` y `alt`. El elemento `<canvas>` tiene solo dos atributos - `width` y `height`. Ambos son opcionales y pueden ser definidos usando propiedades del `DOM`. Cuando los atributos ancho y alto no están especificados, el lienzo se inicializa con 300 pixels de ancho y 150 pixels de alto. El elemento puede ser arbitrariamente re-dimensionado por **CSS**, pero durante el renderizado la imagen es escalada para ajustarse al tamaño de su layout. Si el tamaño del **CSS** no respeta el ratio del canvas inicial, este aparecerá distorsionado.

Nota:

Si su renderizado se ve distorsionado, pruebe especificar los atributos `width` y `height` explícitamente en los atributos del `<canvas>`, y no usando **CSS**.

El atributo `id` no está especificado para el elemento `<canvas>` pero es uno de los atributos globales de HTML el cual puede ser aplicado a cualquier elemento HTML (como `class` por ejemplo). Siempre es buena idea proporcionar un `id` porque esto hace más fácil identificarlo en un `script`.

El elemento `<canvas>` puede ser estilizado como a cualquier imagen normal (`margin`, `border`, `background`, etc). Estas reglas, sin embargo, no afectan a lo dibujado sobre el canvas. Cuando no tenemos reglas de estilo aplicadas al `canvas`, este será completamente transparente.

1.1.1. Contenido alternativo

El elemento `<canvas>` se diferencia de un tag `` en que, como los elementos `<video>`, `<audio>` o `<picture>`, es fácil definir contenido alternativo (fallback content) para mostrarse en navegadores viejos que no soporten el elemento `<canvas>`, como versiones de Internet Explorer previas a la versión 9 o navegadores de texto. Siempre debes proporcionar contenido alternativo para mostrar en estos navegadores.

Proporcionar contenido alternativo es muy explicito: solo debemos insertar el contenido alterno dentro del elemento `<canvas>`. Los navegadores que no soporten `<canvas>` ignorarán el contenido y mostrarán el contenido indicado dentro de este. Y los que soporten `<canvas>` ignorarán el contenido en su interior (de las etiquetas), y mostrarán el `canvas` normalmente.

Por ejemplo, podremos proporcionar un texto descriptivo del contenido del `canvas` o proveer una imagen estática del contenido renderizado. Nos podría quedar algo así:

```
<canvas id="stockGraph" width="150" height="150">
  current stock price: $3.15 +0.15
</canvas>

<canvas id="clock" width="150" height="150">
  
</canvas>
```

1.2. Etiqueta canvas requerida

De manera distinta al elemento ``, el elemento `<canvas>` requiere cerrar la etiqueta con (`</canvas>`).

💡 Nota:

Aunque las versiones anteriores del navegador [Safari](#) de Apple no requería el cierre de la etiqueta, la especificación indica que es necesaria, así que tu deberías incluir esta para asegurarte la compatibilidad. Aquellas versiones de Safari (anteriores versiones a 2.0) renderizarán el contenido de regreso agregándolo al canvas mismo a no ser que utilice trucos de [CSS](#) para enmascararlo. Afortunadamente, los usuarios de aquellas versiones de Safari son raros hoy en día.

Si el contenido alternativo no se necesita, un simple `<canvas id="foo"...></canvas>` es completamente compatible con todos los navegadores que soportan canvas.

1.3. El contexto de renderización

`<canvas>` crea un lienzo de dibujo fijado que expone uno o mas contextos renderizados, los cuales son usados para crear y manipular el contenido mostrado. Nos enfocaremos en renderizacion de contextos 2D. Otros contextos deberán proveer diferentes tipos de renderizaciones; por ejemplo, [WebGL](#) usa un 3D contexto ("experimental-webgl") basado sobre [OpenGL ES](#).

El `canvas` esta inicialmente en blanco. Para mostrar alguna cosa, un `script` primero necesita acceder al contexto a renderizar y dibujar sobre este. El elemento `<canvas>` tiene un method llamado `getContext()`, usado para obtener el contexto a renderizar y sus funciones de dibujo. `getContext()` toma un parámetro, el tipo de contexto. Para graficos 2D, como los que cubre este tutorial, su especificacion es "2d".

```
const canvas = document.getElementById("tutorial");
const ctx = canvas.getContext("2d");
```

La primera linea regresa el nodo `DOM` para el elemento `<canvas>` llamando al método `document.getElementById()`. Una vez tu tienes el elemento nodo, tu puedes acceder al contexto de dibujo usando su método `getContext()`.

1.4. Comprobando soporte

El contenido de regreso que es mostrado en navegadores los cuales no soportan `<canvas>`. Para los `Scripts` puede también comprobarse su soporte desde la programación por un simple test para la presencia del método `getContext()`. Con un trozo de código parecido al que viene debajo:

```
const canvas = document.getElementById("tutorial");

if (canvas.getContext) {
  let ctx = canvas.getContext("2d");
  // drawing code here
} else {
  // canvas-unsupported code here
}
```

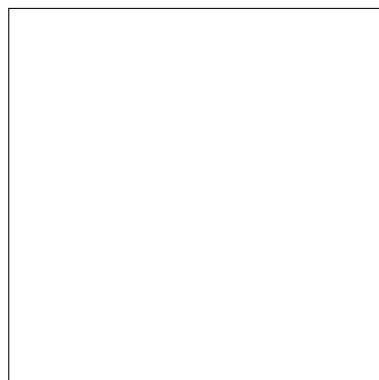
1.5. Un esqueleto de plantilla

Aquí esta una plantilla minimalista, la cual usaremos como punto de partida para posteriores ejemplos.

```
<html>
  <head>
    <title>Canvas tutorial</title>
    <script type="text/javascript">
      function draw() {
        const canvas = document.getElementById("tutorial");
        if (canvas.getContext) {
          let ctx = canvas.getContext("2d");
        }
      }
    </script>
    <style type="text/css">
      canvas {
        border: 1px solid black;
      }
    </style>
  </head>
  <body onload="draw();" >
    <canvas id="tutorial" width="150" height="150"></canvas>
  </body>
</html>
```

El `script` incluye una función llamada `draw()`, la cual es ejecutada una vez finalizada la carga de la pagina; este esta hecho usando el evento `load` del documento. Esta función, o una parecida, podría también ser llamada usando `window.setTimeout()` , `window.setInterval()` , o cualquier otro manejador de evento, a lo largo de que la pagina esta siendo cargada la primera vez.

Así es como se ve el elemento `<canvas>` en la pagina.



1.6. Ejemplo

Para comenzar, daremos un vistazo a un simple ejemplo que dibuja dos rectángulos que se intersecan, uno de los cuales tiene alpha transparencia.

```
<html>
  <head>
    <script type="application/javascript">
      function draw() {
        const canvas = document.getElementById("canvas");
        if (canvas.getContext) {
          let ctx = canvas.getContext("2d");

          ctx.fillStyle = "rgb(200,0,0)";
          ctx.fillRect(10, 10, 55, 50);

          ctx.fillStyle = "rgba(0, 0, 200, 0.5)";
          ctx.fillRect(30, 30, 55, 50);
        }
      }
    </script>
  </head>
  <body onload="draw();" >
    <canvas id="canvas" width="150" height="150"></canvas>
  </body>
</html>
```

Visualmente quedaría así:

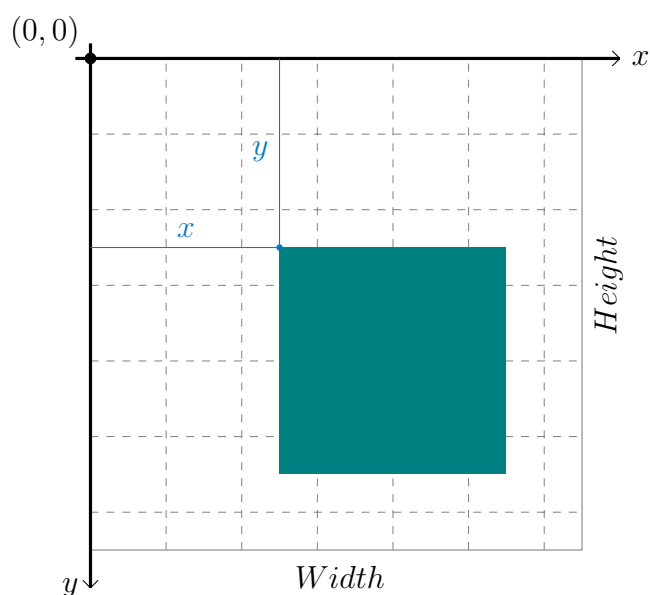


2. Dibujando formas con canvas

Ahora que hemos preparado nuestro entorno `canvas`, podemos entrar en detalles de cómo dibujar en el `canvas`. Al final de esta sección, habrás aprendido cómo dibujar rectángulos, triángulos, líneas, arcos y curvas, familiarizándote con algunas de las formas básicas. Trabajar con trazados es esencial a la hora de dibujar objetos en el `canvas` y veremos cómo hacerlo.

2.1. La cuadrícula

Antes de empezar a dibujar, tenemos que hablar de la cuadrícula del `canvas` o del espacio de coordenadas. Nuestra estructura `HTML` de la sección anterior tenía un elemento de `canvas` de 150 pixels de ancho y 150 pixels de alto.



Normalmente, 1 unidad en la cuadrícula corresponde a 1 pixel en el canvas. El origen de esta cuadrícula se sitúa en la esquina superior izquierda en la `coordenada (0,0)`. Todos los elementos se colocan en relación con este origen. Así que la posición de la esquina superior izquierda del cuadrado azul se sitúa a `X` pixels de la izquierda y a `Y` pixels de la parte superior, en la `coordenada (x,y)`. Más adelante en esta sección veremos cómo podemos trasladar el origen a una posición diferente, rotar la cuadrícula e incluso escalarla, pero por ahora nos ceñiremos a la posición por defecto.

2.2. Dibujar rectángulos

A diferencia de `SVG`, `<canvas>` sólo admite dos formas primitivas: rectángulos y trazados (listas de puntos conectados por líneas). Todas las demás formas deben crearse combinando uno o más trazados. Por suerte, tenemos un surtido de funciones de dibujo de trazados que hacen posible componer formas muy complejas.

Primero veamos el `rectángulo`. Hay tres funciones que dibujan rectángulos en el canvas:

```
// (1) Dibuja un rectangulo relleno.  
fillRect(x, y, width, height)
```

```
// (2) Dibuja un contorno rectangular.  
strokeRect(x, y, width, height) (en-US)
```

```
// (3) Borra el area rectangular especificada, haciendola  
totalmente transparente.  
clearRect(x, y, width, height)
```

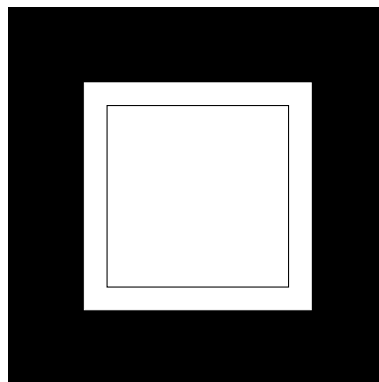
Cada una de estas tres funciones toma los mismos parámetros. `x` y `y` especifican la posición en el `canvas` (relativa al origen) de la esquina superior izquierda del rectángulo. `width` y `height` proporcionan el tamaño del rectángulo.

A continuación se muestra la función `draw()` de la página anterior, pero ahora hace uso de estas tres funciones.

Ejemplo de forma rectangular:

```
function draw() {  
  const canvas = document.getElementById("canvas");  
  if (canvas.getContext) {  
    const ctx = canvas.getContext("2d");  
  
    ctx.fillRect(25, 25, 100, 100);  
    ctx.clearRect(45, 45, 60, 60);  
    ctx.strokeRect(50, 50, 50, 50);  
  }  
}
```

La salida de este ejemplo seria este:



La función `fillRect()` dibuja un gran cuadrado negro de 100 pixels en cada lado. La función `clearRect()` borra un cuadrado de 60x60 pixels del centro, y luego se llama a `strokeRect()` para crear un contorno rectangular de 50x50 pixels dentro del cuadrado borrado.

En las próximas páginas veremos dos métodos alternativos para `clearRect()`, y también veremos cómo cambiar el color y el estilo de trazo de las formas renderizadas.

A diferencia de las funciones de trazado que veremos en la siguiente sub-sección, las tres funciones de rectángulo dibujan inmediatamente en el canvas.

2.3. Dibujando paths

Veamos ahora los `paths` (trazos). Un `paths` es una lista de puntos, conectados por segmentos de líneas que pueden ser de diferentes formas, curvas o no; de diferente anchura y de diferente color. Un `path`, o incluso un sub-path, puede ser cerrado. Para hacer formas usando trazos, damos algunos pasos adicionales:

1. Primero, se crea el path o trazo.
2. Luego, se utiliza comandos de dibujo para dibujar en el path.
3. Una vez creado el path, puedes trazar o rellenar el path para renderizarlo.

Aquí están las funciones utilizadas para realizar estos pasos:

- **`beginPath()`**: Crea un nuevo trazado. Una vez creado, los futuros comandos de dibujo se dirigen al trazado y se utilizan para construirlo.
- **Métodos de `path`**: Métodos para establecer diferentes paths para los objetos.
- **`closePath()`**: Añade una línea recta al path, que va al inicio del sub-path actual.
- **`stroke()`**: Dibuja la forma trazando su contorno.
- **`fill()`**: Dibuja una forma sólida rellenando el área de contenido del trazo.

El primer paso para crear un trazo(`path`) es llamar a `beginPath()`. Internamente, los trazos se almacenan como una lista de sub-trazos (líneas, arcos, etc.) que juntos forman una forma. Cada vez que se llama a este método, la lista se restablece y podemos empezar a dibujar nuevas formas.

💡 Nota:

Cuando el trazo actual está vacío, como por ejemplo inmediatamente después de llamar a `beginPath()`, o en un canvas recién creado, el primer comando de construcción del trazo siempre se trata como un `moveTo()`, independientemente de lo que realmente sea. Por esta razón, casi siempre querrá establecer específicamente su posición inicial después de reiniciar un trazo.

El segundo paso es llamar a los métodos que realmente especifican los trazos a dibujar. Los veremos en breve.

El tercer paso, y opcional, es llamar a `closePath()`. Este método intenta cerrar la forma dibujando una línea recta desde el punto actual hasta el inicio. Si la forma ya ha sido cerrada o sólo hay un punto en la lista, esta función no hace nada.

💡 Nota:

Cuando se llama a `fill()`, cualquier forma abierta se cierra automáticamente, por lo que no es necesario llamar a `closePath()`. Este no es el caso cuando se llama a `stroke()`.

2.3.1. Dibujar un triángulo

Por ejemplo, el código para dibujar un triángulo sería algo así:

```
function draw() {  
  const canvas = document.getElementById("canvas");  
  if (canvas.getContext) {  
    const ctx = canvas.getContext("2d");  
  
    ctx.beginPath();  
    ctx.moveTo(75, 50);  
    ctx.lineTo(100, 75);  
    ctx.lineTo(100, 25);  
    ctx.fill();  
  }  
}
```

El resultado se ve así:



2.3.2. Dibujando con el lápiz

Una función muy útil, que en realidad no dibuja nada sino que se convierte en parte de la lista de trazos descrita anteriormente, es la función `moveTo()`. La mejor manera de pensar en esto es como [levantar un bolígrafo o un lápiz](#) de un lugar en un pedazo de papel y colocarlo en el siguiente.

- **`moveTo(x, y)`:** Muévete como usando el lápiz, por las coordenadas `x` e `y`.

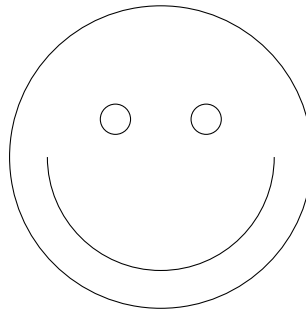
Cuando se inicializa el canvas o se llama a `beginPath()`, normalmente se querrá utilizar la función `moveTo()` para colocar el punto de partida en otro lugar. También podemos usar `moveTo()` para dibujar trazos no conectados. Echa un vistazo a la cara sonriente de abajo.

Para probarlo por ti mismo, puedes utilizar el siguiente fragmento de código. Sólo tienes que pegarlo en la función `draw()` que vimos antes.

```
function draw() {
  const canvas = document.getElementById("canvas");
  if (canvas.getContext) {
    const ctx = canvas.getContext("2d");

    ctx.beginPath();
    ctx.arc(75, 75, 50, 0, Math.PI * 2, true); // Circulo externo
    ctx.moveTo(110, 75);
    ctx.arc(75, 75, 35, 0, Math.PI, false); // Boca (en el
    sentido de las agujas del reloj)
    ctx.moveTo(65, 65);
    ctx.arc(60, 65, 5, 0, Math.PI * 2, true); // Ojo izquierdo
    ctx.moveTo(95, 65);
    ctx.arc(90, 65, 5, 0, Math.PI * 2, true); // Ojo derecho
    ctx.stroke();
  }
}
```

El resultado se ve así:



Si quisieras ver las líneas conectadas, puedes eliminar las líneas que llaman a `moveTo()`.

2.3.3. Líneas

Para dibujar líneas rectas, utilice el método `lineTo()`.

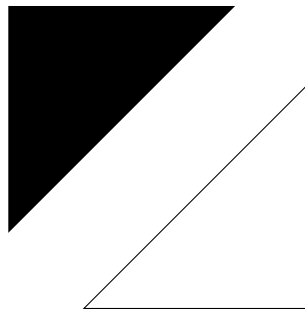
- **lineTo(x, y):** Dibuja una línea desde la posición actual de dibujo hasta la posición especificada por `x` e `y`.

Este método toma dos argumentos, `x` e `y`, que son las coordenadas del punto final de la línea. El punto de partida depende de los trazos anteriores, donde el punto final del trazo anterior es el punto de partida del siguiente, etc. El punto de partida también puede cambiarse utilizando el método `moveTo()`.

El ejemplo siguiente dibuja dos triángulos, uno relleno y otro contorneado.

```
function draw() {  
  const canvas = document.getElementById("canvas");  
  if (canvas.getContext) {  
    const ctx = canvas.getContext("2d");  
  
    // Triangulo relleno  
    ctx.beginPath();  
    ctx.moveTo(25, 25);  
    ctx.lineTo(105, 25);  
    ctx.lineTo(25, 105);  
    ctx.fill();  
  
    // Triangulo contorneado  
    ctx.beginPath();  
    ctx.moveTo(125, 125);  
    ctx.lineTo(125, 45);  
    ctx.lineTo(45, 125);  
    ctx.closePath();  
    ctx.stroke();  
  }  
}
```

Esto comienza llamando a `beginPath()` para iniciar un nuevo trazo. A continuación, utilizamos el método `moveTo()` para mover el punto de partida a la posición deseada. Debajo de esto, se dibujan dos líneas que forman los dos lados del triángulo.



Notará la diferencia entre el triángulo relleno y el trazo. Esto se debe, como se ha mencionado anteriormente, a que las formas se cierran automáticamente cuando se rellena un trazo, pero no cuando se traza. Si omitimos el `closePath()` para el triángulo trazo, sólo se habrían dibujado dos líneas, no un triángulo completo.

2.3.4. Arcos

Para dibujar arcos o círculos, utilizamos los métodos `arc()` o `arcTo()`.

- **`arc(x, y, radius, startAngle, endAngle, anticlockwise)`**: Dibuja un arco centrado en la posición `(x, y)` con radio `r` que comienza en `startAngle` y termina en `endAngle` yendo en la dirección indicada por `counterclockwise` (por defecto en el sentido de las agujas del reloj).
- **`arcTo(x1, y1, x2, y2, radius)`**: Dibuja un arco con los puntos de control y el radio dados, conectado al punto anterior por una línea recta.

Veamos con más detalle el método `arc`, que toma seis parámetros: `x` e `y` son las coordenadas del centro del círculo sobre el que se dibujará el arco. El parámetro radio se explica por sí mismo. Los parámetros `startAngle` y `endAngle` definen los puntos inicial y final del arco en radianes, a lo largo de la curva del círculo. Se miden desde el eje `x`. El parámetro `counterclockwise` es un valor `Booleano` que, cuando es `true`, dibuja el arco en sentido contrario a las agujas del reloj; en caso contrario, el arco se dibuja en sentido de las agujas del reloj.

💡 Nota:

Los ángulos en la función `arc()` se miden en radianes, no en grados. Para convertir los grados en radianes puedes utilizar la siguiente expresión de **JavaScript**:

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

El siguiente ejemplo es un poco más complejo que los que hemos visto anteriormente. Dibuja 12 arcos diferentes, todos con diferentes ángulos y rellenos.

Los dos bucles `for` son para recorrer las filas y columnas de arcos. Para cada arco, iniciamos un nuevo trazo llamando a `beginPath()`. En el código, cada uno de los parámetros del arco está en una variable para mayor claridad, pero no necesariamente se haría eso en la vida real.

Las coordenadas `x` e `y` deberían ser lo suficientemente claras. `radius` y `startAngle` son fijos. `endAngle` comienza en 180 grados (medio círculo) en la primera columna y se incrementa en pasos de 90 grados, culminando en un círculo completo en la última columna.

La sentencia para el parámetro `clockwise` hace que la primera y tercera fila se dibujen como arcos en el sentido de las agujas del reloj y la segunda y cuarta fila como arcos en sentido contrario. Por último, la sentencia `if` hace que la mitad superior tenga arcos trazados y la mitad inferior arcos rellenos.

💡 Nota:

Este ejemplo requiere un `canvas` ligeramente más grande que los otros de esta página: 150 x 200 pixels.

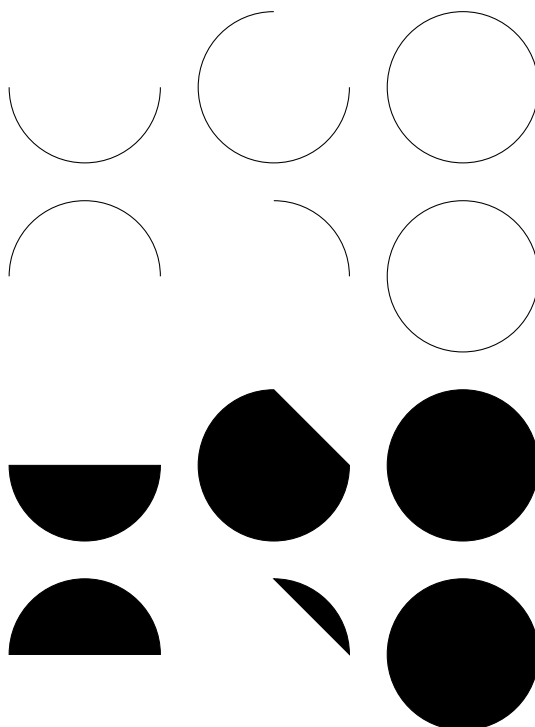
```
function draw() {
  const canvas = document.getElementById("canvas");
  if (canvas.getContext) {
    const ctx = canvas.getContext("2d");

    for (let i = 0; i < 4; i++) {
      for (let j = 0; j < 3; j++) {
        ctx.beginPath();
        const x = 25 + j * 50; // Coordenada x
        const y = 25 + i * 50; // Coordenada y
        const radius = 20; // Radio del Arco
        const startAngle = 0; // Punto inicial del Circulo
        const endAngle = Math.PI + (Math.PI * j) / 2; // Punto
        final del Circulo
        const counterclockwise = i % 2 !== 0; // En el sentido
        de las agujas del reloj o en sentido contrario

        ctx.arc(x, y, radius, startAngle, endAngle,
        counterclockwise);

        if (i > 1) {
          ctx.fill();
        } else {
          ctx.stroke();
        }
      }
    }
  }
}
```

El resultado de todo ese código es el siguiente:

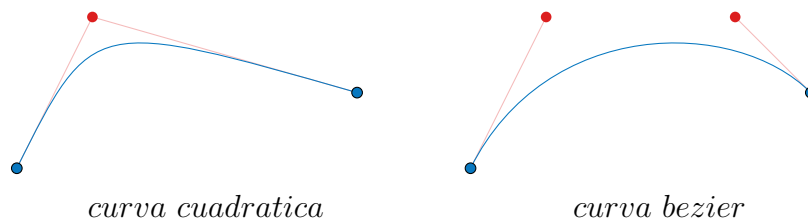


2.3.5. Curvas de Bézier y cuadráticas

El siguiente tipo de trayectorias disponibles son las Curvas de Bézier, disponibles en las variedades cúbica y cuadrática. Se utilizan generalmente para dibujar formas orgánicas complejas.

- **quadraticCurveTo(cp1x, cp1y, x, y):** Dibuja una curva cuadrática de Bézier desde la posición actual de la pluma hasta el punto final especificado por `x` e `y`, utilizando el punto de control especificado por `cp1x` y `cp1y`.
- **bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y):** Dibuja una curva cúbica de Bézier desde la posición actual de la pluma hasta el punto final especificado por `x` e `y`, utilizando los puntos de control especificados por `(cp1x, cp1y)` y `(cp2x, cp2y)`.

La diferencia entre ellas es que una curva de Bézier cuadrática tiene un punto inicial y otro final (puntos azules) y sólo un punto de control (indicado por el punto rojo) mientras que una curva de Bézier cúbica utiliza dos puntos de control.



Los parámetros `x` e `y` de estos dos métodos son las coordenadas del punto final. Los parámetros `cp1x` y `cp1y` son las coordenadas del primer punto de control, y `cp2x` y `cp2y` son las coordenadas del segundo punto de control.

El uso de las curvas cuadráticas y cúbicas de Bézier puede ser un reto, porque a diferencia de los programas de dibujo vectorial como Adobe Illustrator, no tenemos información visual directa sobre lo que estamos haciendo. Esto hace que sea bastante difícil dibujar formas complejas. En el siguiente ejemplo, dibujaremos algunas formas orgánicas simples, pero si tienes tiempo y, sobre todo, paciencia, se pueden crear formas mucho más complejas.

No hay nada muy difícil en estos ejemplos. En ambos casos vemos cómo se dibuja una sucesión de curvas que finalmente dan lugar a una forma completa.

Curvas cuadráticas de Bézier

Este ejemplo utiliza múltiples curvas cuadráticas de Bézier para representar un comentario de voz.

```
function draw() {  
  const canvas = document.getElementById("canvas");  
  if (canvas.getContext) {  
    const ctx = canvas.getContext("2d");  
  
    // Ejemplo de curvas cuadráticas  
    ctx.beginPath();  
    ctx.moveTo(75, 25);  
    ctx.quadraticCurveTo(25, 25, 25, 62.5);  
    ctx.quadraticCurveTo(25, 100, 50, 100);  
    ctx.quadraticCurveTo(50, 120, 30, 125);  
    ctx.quadraticCurveTo(60, 120, 65, 100);  
    ctx.quadraticCurveTo(125, 100, 125, 62.5);  
    ctx.quadraticCurveTo(125, 25, 75, 25);  
    ctx.stroke();  
  }  
}
```

El resultado se ve así:

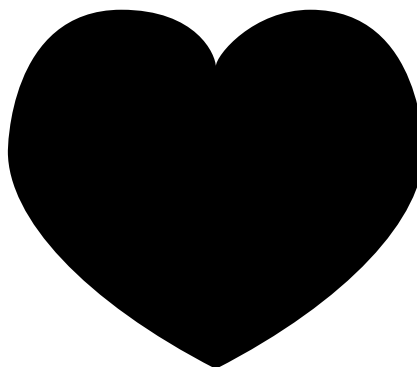


Curvas cúbicas de Bézier

Este ejemplo dibuja un corazón utilizando curvas cúbicas de Bézier.

```
function draw() {  
  const canvas = document.getElementById("canvas");  
  if (canvas.getContext) {  
    const ctx = canvas.getContext("2d");  
  
    // Ejemplo de curvas cubicas  
    ctx.beginPath();  
    ctx.moveTo(75, 40);  
    ctx.bezierCurveTo(75, 37, 70, 25, 50, 25);  
    ctx.bezierCurveTo(20, 25, 20, 62.5, 20, 62.5);  
    ctx.bezierCurveTo(20, 80, 40, 102, 75, 120);  
    ctx.bezierCurveTo(110, 102, 130, 80, 130, 62.5);  
    ctx.bezierCurveTo(130, 62.5, 130, 25, 100, 25);  
    ctx.bezierCurveTo(85, 25, 75, 37, 75, 40);  
    ctx.fill();  
  }  
}
```

El resultado se ve así:



2.3.6. Rectángulos

Además de los tres métodos que vimos en [Dibujar rectángulos \(sub-sección 2.2\)](#), que dibujan formas rectangulares directamente en el `canvas`, existe también el método `rect()`, que añade un trazado rectangular a un trazado actualmente abierto.

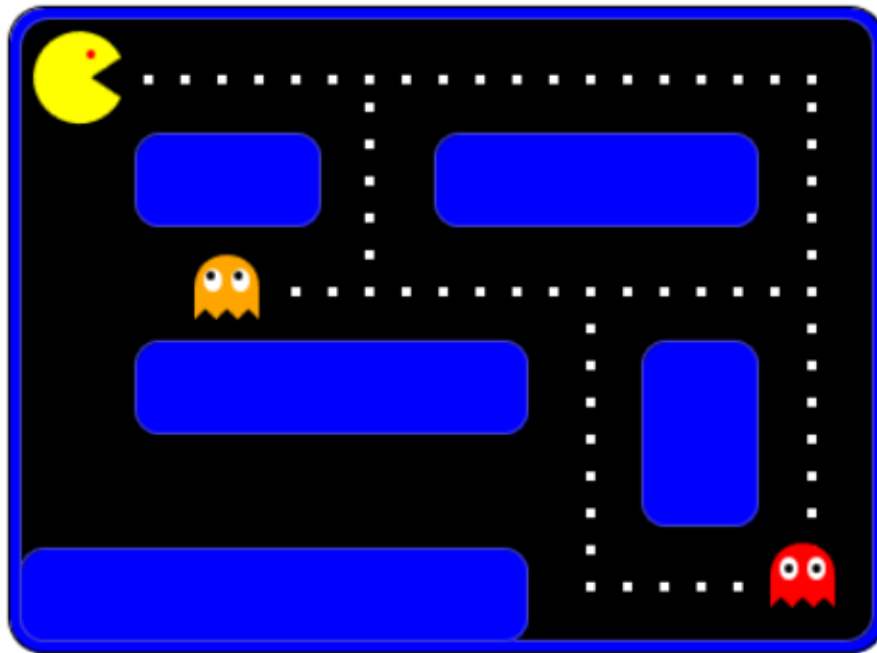
- **rect(x, y, width, height):** Dibuja un rectángulo cuya esquina superior izquierda está especificada por (x, y) con el `width` y `height` especificados.

Antes de que se ejecute este método, se llama automáticamente al método `moveTo()` con los parámetros (x, y). En otras palabras, la posición actual de la pluma se restablece automáticamente a las coordenadas por defecto.

2.3.7. Hacer combinaciones

Hasta ahora, cada ejemplo de esta página ha utilizado sólo un tipo de función de trazado por forma. Sin embargo, no hay ninguna limitación en cuanto al número o los tipos de trazos que puedes utilizar para crear una forma. Así que en este último ejemplo, vamos a combinar todas las funciones de trazado para recrear el famoso juego de **Pacman**.

Para los mas ansiosos, el resultado final seria este:



Iniciamos este proyecto con el código **HTML**.

```
<html>
  <body onload="draw();">
    <canvas id="canvas" width="400" height="300"></canvas>
  </body>
</html>
```

Dentro de la etiqueta `<body>` usamos el atributo `onload` con el valor `draw()`. Esto para que se ejecute lo que esta dentro de la funcion `draw()` dentro de **JavaScript**.

Y para la etiqueta `<canvas>` usamos el atributo `id` con el valor `canvas` y el atributo `width` con el valor `400` y `height` con el valor `300`. Esto significa que se creara un area de `400 x 300` para colocar tantos trazos como queramos.

El `id` con valor `"canvas"`, se usa para poder hacer referencia a la etiqueta `<canvas>` dentro del **JavaScript**.

Ahora el código en **JavaScript**

```
function draw() {
  const canvas = document.getElementById('canvas');

  if (canvas.getContext) {
    const ctx = canvas.getContext('2d');

    roundedRect(ctx, 10, 10, 380, 280, 15);
    roundedRect(ctx, 15, 15, 370, 270, 12, 'black', true);

    // bloques
    roundedRect(ctx, 65, 65, 80, 40, 10);
    roundedRect(ctx, 195, 65, 140, 40, 10);
    roundedRect(ctx, 65, 155, 170, 40, 10);
    roundedRect(ctx, 15, 245, 220, 40, 10);
    roundedRect(ctx, 285, 155, 50, 80, 10);

    // dibujando a Pacman
    bodyPacman(ctx);
    pupila(ctx, 45, 30, 'red');

    // ruta para alcanzar la comida
    horizontalRoute(ctx, 68, 39, 19);
    horizontalRoute(ctx, 132, 131, 15);
    horizontalRoute(ctx, 260, 259, 5);
    verticalRoute(ctx, 164, 51, 5);
    verticalRoute(ctx, 356, 51, 13);
    verticalRoute(ctx, 260, 131, 9);

    // cuerpo del fantasma naranja
    bodyPhantom(ctx, 90, 145);
    eyes(ctx, 98, 123);
    pupila(ctx, 97, 126);
    pupila(ctx, 109, 126);

    // cuerpo del fantasma rojo
    bodyPhantom(ctx, 340, 270, 'red');
    eyes(ctx, 348, 248);
    pupila(ctx, 348, 253);
    pupila(ctx, 360, 253);
  }
}
```

Creamos funciones auxiliares para dibujar las formas que vamos a utilizar. Hacemos esto para mantener el código mas ordenado, por legibilidad y para evitar la redundancia.

- **roundedRect():** Función para dibujar rectángulos redondeados.
- **bodyPacman():** Función para dibujar el cuerpo del Pacman.
- **pupila():** Función para dibujar la pupila de los personajes.
- **bodyPhantom():** Función para dibujar el cuerpo de los fantasmas.
- **eyes():** Función para dibujar los ojos de los fantasmas.

- **horizontalRoute()**: Función para dibujar la ruta horizontal.
- **verticalRoute()**: Función para dibujar la ruta vertical.

A todas las funciones auxiliares, les pasamos atributos como `ctx`, `x`, `y` y `color`. Como los mas comunes, en el caso de `ctx` es el `canvas.getContext("2d")`. Solo es una forma de reciclar código.

La function `roundedRect()` dibuja un rectángulo redondeado. Entre los aspectos que se destaca, son el color de `fillStyle`; por defecto usa el color `blue`, pero si usas el parámetro `color`, puedes reemplazarlo por otro.

```
const roundedRect = (
  ctx,
  x,
  y,
  width,
  height,
  r,
  color = 'blue',
  stroke = false
) => {
  if (stroke) ctx.strokeStyle = '#4141CC';

  ctx.fillStyle = color;
  ctx.beginPath();
  ctx.moveTo(x, y + r);
  ctx.arcTo(x, y + height, x + r, y + height, r);
  ctx.arcTo(x + width, y + height, x + width, y + height - r, r);
  ctx.arcTo(x + width, y, x + width - r, y, r);
  ctx.arcTo(x, y, x, y + r, r);
  ctx.fill();
  ctx.stroke();
};
```

Por otro lado tenemos a la función `bodyPacman()` que dibuja el cuerpo de Pacman. El código es simple, con `fillStyle` igual a `"yellow"` se establece el color de relleno, los primeros 2 parámetros de `ctx.arc()` definen su posición y el tercero el radio que tendrá.

```
const bodyPacman = (ctx) => {
  ctx.fillStyle = 'yellow';
  ctx.beginPath();
  ctx.arc(40, 40, 20, Math.PI / 7, -Math.PI / 7, false);
  ctx.lineTo(45, 40);
  ctx.fill();
};
```

Para la función `bodyPhantom()` que es el cuerpo de los fantasmas, usa un color por defecto “orange” y también se puede cambiar si pasas un nuevo parámetro de color. Puedes crear tantos fantasmas como quieras, solo tienes que definir su posición con los ejes `x` y `y`.

```
const bodyPhantom = (ctx, x, y, color = 'orange') => {
  ctx.fillStyle = color;
  ctx.beginPath();
  ctx.moveTo(x, y);
  ctx.lineTo(x, y - 14);
  ctx.bezierCurveTo(x, y - 22, x + 6, y - 28, x + 14, y - 28);
  ctx.bezierCurveTo(x + 22, y - 28, x + 28, y - 22, x + 28, y - 14);
  ctx.lineTo(x + 28, y);
  ctx.lineTo(x + 23.333, y - 4.667);
  ctx.lineTo(x + 18.666, y);
  ctx.lineTo(x + 14, y - 4.667);
  ctx.lineTo(x + 9.333, y);
  ctx.lineTo(x + 4.666, y - 4.667);
  ctx.lineTo(x, y);
  ctx.fill();
};
```

Las funciones `pupila()` y `eyes()` dibujan las pupilas y los ojos de los personajes. Los color de relleno son “black” y “white” correspondientemente. Todas las funciones se llaman usando los ejes de posición `x` y `y`, para evitar calcular toda la lógica que esto implica.

```
const pupila = function (ctx, x, y, color = 'black') {
  ctx.fillStyle = color;
  ctx.beginPath();
  ctx.arc(x, y, 2, 0, Math.PI * 2, true);
  ctx.fill();
};

const eyes = function (ctx, x, y, color = 'white') {
  ctx.fillStyle = color;
  ctx.beginPath();
  ctx.moveTo(x, y);
  ctx.bezierCurveTo(x - 3, y, x - 4, y + 3, x - 4, y + 5);
  ctx.bezierCurveTo(x - 4, y + 7, x - 3, y + 10, x, y + 10);
  ctx.bezierCurveTo(x + 3, y + 10, x + 4, y + 7, x + 4, y + 5);
  ctx.bezierCurveTo(x + 4, y + 3, x + 3, y, x, y);
  ctx.moveTo(x + 12, y);
  ctx.bezierCurveTo(x + 9, y, x + 8, y + 3, x + 8, y + 5);
  ctx.bezierCurveTo(x + 8, y + 7, x + 9, y + 10, x + 12, y + 10);
  ctx.bezierCurveTo(x + 15, y + 10, x + 16, y + 7, x + 16, y + 5);
  ctx.bezierCurveTo(x + 16, y + 3, x + 15, y, x + 12, y);
  ctx.fill();
};
```

Para finalizar tenemos las funciones `horizontalRoute()` y `verticalRoute()` que dibujan las rutas horizontales y verticales. El color de relleno es "white", los parámetros `x` y `y` indican su posición inicial y el parámetro `amount` la cantidad de veces que se repetirá con una separación de 16pt la una de la otra.

```
const horizontalRoute = function (ctx, x, y, amount) {
  ctx.fillStyle = 'white';
  for (let i = 0; i < amount; i++) {
    ctx.fillRect(x + i * 16, y, 4, 4);
  }
};

const verticalRoute = function (ctx, x, y, amount) {
  ctx.fillStyle = 'white';
  for (i = 0; i < amount; i++) {
    ctx.fillRect(x, y + i * 16, 4, 4);
  }
};
```

*Fin del juego de **Pacman***

2.4. Objetos Path2D

Como hemos visto en el último ejemplo, puede haber una serie trazos y comandos de dibujo para dibujar objetos en su `canvas`. Para simplificar el código y mejorar el rendimiento, el objeto `Path2D`, disponible en las versiones recientes de los navegadores, le permite almacenar en caché o grabar estos comandos de dibujo. De este modo, se pueden reproducir los trazos rápidamente. Veamos cómo podemos construir un objeto `Path2D`:

- **Path2D():** El constructor `Path2D()` devuelve un objeto `Path2D` recién instanciado, opcionalmente con otra ruta como argumento (crea una copia), u opcionalmente con una cadena de caracteres formada por datos de un trazo `SVG path`.

```
new Path2D(); // Objeto Path2D vacío
new Path2D(path); // Copia de otro objeto Path2D
new Path2D(d); // Path2D a partir de un trazo (SVG path)
```

Todos los Métodos de trazo como `moveTo`, `rect`, `arc` o `quadraticCurveTo`, etc., que hemos conocido anteriormente, están disponibles en los objetos `Path2D`.

La API `Path2D` también añade una forma de combinar trazados mediante el método `addPath`. Esto puede ser útil cuando se quiere construir objetos a partir de varios componentes, por ejemplo.

- **Path2D.addPath(path [, transform]):** Añade un trazo al trazo actual con una matriz de transformación opcional.

2.4.1. Ejemplo de Path2D

En este ejemplo, estamos creando un rectángulo y un círculo. Ambos se almacenan como un objeto `Path2D`, para que estén disponibles para su uso posterior. Con la nueva [API Path2D](#), varios métodos se han actualizado para aceptar opcionalmente un objeto `Path2D` para utilizarlo en lugar del trazo actual. Aquí, `stroke` y `fill` se utilizan con un argumento de trazo para dibujar ambos objetos en el `canvas`, por ejemplo.

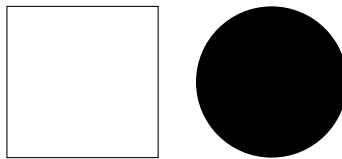
```
function draw() {
  const canvas = document.getElementById("canvas");
  if (canvas.getContext) {
    const ctx = canvas.getContext("2d");

    const rectangle = new Path2D();
    rectangle.rect(10, 10, 50, 50);

    const circle = new Path2D();
    circle.arc(100, 35, 25, 0, 2 * Math.PI);

    ctx.stroke(rectangle);
    ctx.fill(circle);
  }
}
```

El resultado se vería así.



2.4.2. Uso de trazados (SVG paths)

Otra poderosa característica de la nueva [API Path2D](#) del `canvas` es el uso de datos de trazados o `SVG path` para inicializar los trazos en el `canvas`. Esto podría permitirle pasar los datos del trazo y reutilizarlos tanto en el [SVG](#) como en el `canvas`.

El trazo se moverá al punto (`M10 10`) y luego se moverá horizontalmente `80` puntos a la derecha (`h 80`), luego `80` puntos hacia abajo (`v 80`), luego `80` puntos a la izquierda (`h -80`), y luego de vuelta al inicio (`z`).

Finalizaremos la sección [\(2.2\) Dibujando formas con canvas](#) con este ultimo ejemplo.

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

let p = new Path2D("M10 10 h 80 v 80 h -80 Z");
ctx.fill(p);
```

