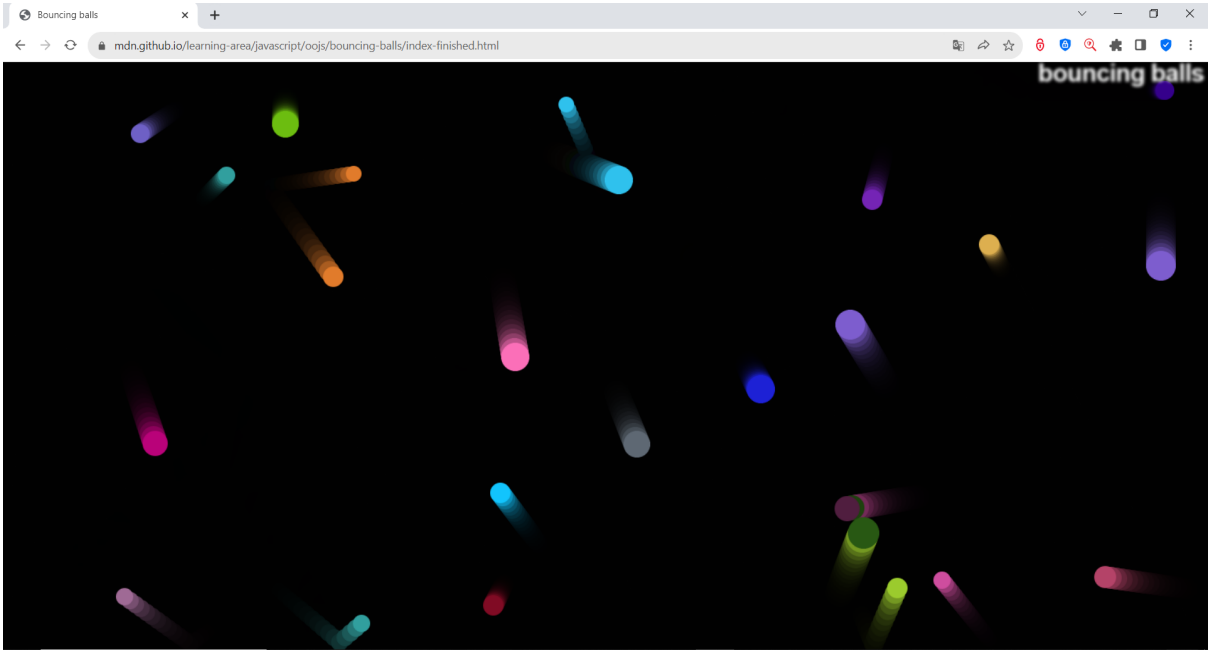
Práctica en JavaScript: Construcción y manejo de Objetos

Mediante esta práctica se pretende mejorar la comprensión y manejo de los objetos definidos por el usuario en JavaScript. El objetivo es crear una aplicación como demo del juego clásico de pelotas que rebotan. El resultado final será semejante al siguiente:



Durante esta práctica se utilizará Canvas API para representar las pelotas en la pantalla, así como el método window.requestAnimationFrame con la finalidad de generar la animación. Así, durante la realización de la práctica se usarán objetos y técnicas que permitan configurar el comportamiento de rebote de las pelotas, tanto en los bordes de la pantalla como cuando interactúen entre ellas.

Paso 1: Creación de los archivos HTML y CSS.

El documento HTML simplemente va a contener un encabezado y el elemento <canvas> en el que se animará el comportamiento de las pelotas.

```html
<!DOCTYPE html>
<html lang="en-us">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>Bouncing balls</title>
    <link rel="stylesheet" href="balls.css">
  </head>

  <body>
    <h1>bouncing balls</h1>
    <canvas></canvas>

    <script src="balls.js"></script>
  </body>
</html>
```

El archivo CSS permitirá localizar el encabezado h1, así como ocultar las barras de desplazamiento y los bordes de la página:

```css
html, body {
  margin: 0;
}
html {
  font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
  height: 100%;
}
body {
  overflow: hidden;
  height: inherit;
}
h1 {
  font-size: 2rem;
  letter-spacing: -1px;
  position: absolute;
  margin: 0;
  top: -4px;
  right: 5px;

  color: transparent;
  text-shadow: 0 0 4px white;
}
```

Paso 2: Creación del archivo js y configuración básica.

En primer lugar, vamos a realizar el *Set-up* del canvas:

```
//Set-up del canvas

let canvas = document.getElementsByTagName('canvas')[0];
let ctx = canvas.getContext('2d');

let width = canvas.width = window.innerWidth;
let height = canvas.height = window.innerHeight;
```

Este script obtiene el elemento <canvas> y luego llama al método getContext('2d') dando lugar a la creación de un objeto que representa un contexto de renderizado de dos dimensiones en el área del dibujo del <canvas>. A continuación, da valor a las variables width y height que corresponden al ancho y alto del elemento canvas (**canvas.width y canvas.height**), de manera que el alto y ancho coincidan con el alto y ancho del navegador (**window.innerWidth y window.innerHeight**). Tal y como se observa, la asignación se ha hecho de forma directa encadenándolas.

Por último, crearemos una función que genere números aleatorios entre unos valores máximos y mínimos que posteriormente servirá para la animación de las pelotas:

```
//Función para generar números aleatorios
function random(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Paso 3: modelado de las pelotas del programa.

Puesto que en este programa tendremos gran cantidad de pelotas rebotando por toda la pantalla y todas tienen el mismo comportamiento, lo lógico en este caso es recurrir a la creación de un objeto[1]. Empezamos definiendo un constructor para el objeto pelota (Ball), en nuestro código:

```
function Ball(x, y, velX, velY, color, size) {
  this.x = x; //posición horizontal
  this.y = y; //posición vertical
  this.velX = velX; //velocidad horizontal
  this.velY = velY; //velocidad vertical
  this.color = color; //color
  this.size = size; //tamaño
}
```

[1] Igualmente, y en el manejo de este tipo de programas se pueden crear una clase para este tipo de objetos.

Aquí se han incluido algunos parámetros que serán las propiedades de cada pelota:

- **Coordenadas x e y:** pueden variar entre un valor 0 (el la esquina superior izquierda) hasta el valor del ancho y alto del navegador (esquina inferior derecha).
- **Velocidad horizontal y vertical (velX y velY):** estos valores se añadirán a las coordenadas x e y cuando animemos el movimiento de las pelotas, así en cada incremento de visualización de frame, se desplazarán esta cantidad.
- **Color:** cada pelota posee un color.
- **Size:** cada pelota tiene un tamaño (radio) en píxeles.

Paso 4: representación de las pelotas. Añadiendo métodos al objeto.

Para dibujar, añadiremos el método **draw()** al prototipo del objeto **Ball()** mediante la siguiente instancia[2]:

```
23 ▽ Ball.prototype.draw = function () {
24     ctx.beginPath();
25     ctx.fillStyle = this.color;
26     ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);
27     ctx.fill();
28   };
```

Con esta función cada objeto pelota Ball() puede dibujarse en la pantalla utilizando el contexto 2D definido anteriormente en el canvas (ctx).

- beginPath() para declarar que empezaremos a dibujar una forma en el canvas. Esta API de Canvas 2D comienza una nueva ruta.
- fillStyle permite definir el color de la forma. En este caso haremos que coincida con el color definido para el objeto ball.
- arc(x, y, radio, startAngle, endAngle) es un método de la API de Canvas 2D que añade un arco a la trayectoria centrada en la posición (x, y), con el radio r comenzando en startAngle y terminando en endAngle (en radianes). La posición x e y del centro del arco corresponden al centro de la pelota. Los últimos dos parámetros especifican el comienzo y final del arco en radianes. En este caso se especifican 0 y 2π que corresponden a 0 y 360 grados que permite delimitar un círculo completo.
- fill() es un método que finaliza el dibujo y rellena el área de la curva especificada según se indicó con el fillStyle.

[2] Aunque los métodos se pueden definir de forma directa para los objetos cuando se definen también se pueden añadir a un objeto ya definido mediante la instancia **objeto.prototype.nombremetodo = function() {//código de la función a ejecutar}**

Paso 5: testeo de la aplicación hasta este punto.

Como paso previo a la animación de las pelotas vamos a testar la aplicación hasta este punto para asegurarnos de que representa correctamente las mismas en el canvas. Para ello:

- Cargue el archivo HTML en un navegador.
- Abra la consola de JavaScript en el navegador, y refresque la página, para que el tamaño del canvas modifique sus dimensiones adaptándose al viewport con la consola abierta.
- Cree en la consola una nueva pelota como, por ejemplo:

let testBall = new Ball(50, 100, 4, 4, "blue", 10);

- Ejecute el método draw() para dibujar testBall en el canvas (**testBall.draw()**)

Al teclear esta última línea en la consola debería ver representada la pelota en el canvas.

Paso 6: Animando las pelotas.

Ahora que ya podemos dibujar una pelota en una posición dada vamos a empezar a moverla para lo que necesitamos crear una función que actualice la misma. Para ello, creamos un método **update()** que añadimos al prototipo del objeto **Ball** mediante el siguiente código:

```
Ball.prototype.update = function () {
  if (this.x + this.size >= width) {
    this.velX = -this.velX;
  }

  if (this.x - this.size <= 0) {
    this.velX = -this.velX;
  }

  if (this.y + this.size >= height) {
    this.velY = -this.velY;
  }

  if (this.y - this.size <= 0) {
    this.velY = -this.velY;
  }

  this.x += this.velX;
  this.y += this.velY;
}
```

Las cuatro primeras líneas de la función verifican si la pelota **ha alcanzado el borde del canvas**.En caso afirmativo, se invierte la dirección de la velocidad de modo que la misma se desplace en la dirección contraria. Así mismo, se ha tenido en cuenta el tamaño (size) de la pelota en los cálculos puesto que las coordenadas x e y se corresponden con el centro de la misma y lo que se busca es el efecto de que las pelotas rebotan cuando alcanzan el borde del canvas.

Por último, las dos últimas líneas de este método suman las velocidades x e y en sus correspondientes coordenadas, de modo que se logra el efecto de movimiento de las mismas cada vez que este método es invocado.

A continuación vamos a comenzar a crear y animar las pelotas. Para ello vamos a escribir un bloque de código que permita crear pelotas, guardarlas y mediante un comportamiento iterativo cree y actualice constantemente la posición de las mismas:

```
let balls = [];

function loop() {
  ctx.fillStyle = "rgba(0, 0, 0, 0.25)";
  ctx.fillRect(0, 0, width, height);

  while (balls.length < 25) {
    let size = random(10, 20);
    let ball = new Ball(
      // la posición de las pelotas, se dibujará al menos siempre
      // como mínimo a un ancho de la pelota de distancia al borde del
      // canvas, para evitar errores en el dibujo
      random(0 + size, width - size),
      random(0 + size, height - size),
      random(-7, 7),
      random(-7, 7),
      `rgb(${random(0, 255)},${random(0, 255)},${random(0, 255)})`,
      size,
    );
    balls.push(ball);
  }

  for (let i = 0; i < balls.length; i++) {
    balls[i].draw();
    balls[i].update();
  }
  requestAnimationFrame(loop);
}
```

Este bloque de código se encarga de:

- Generar un array para almacenar las pelotas creadas.

- Definir el color de relleno del canvas como negro semi-transparente y dibujar un rectángulo en todo el ancho y alto del canvas, usando **fillRect()** (los cuatro parámetros definen las coordenadas de origen, el ancho y el alto del rectángulo). Esto es para cubrir el dibujo del instante anterior antes de actualizar el nuevo dibujo. Si no se realiza este paso, resultará en las imágenes se irán apilando y veremos una especie de serpientes según se mueven por el canvas en vez de las pelotas moviéndose. El color de relleno se define como semitransparente, rgba(0,0,0,0.25) lo que nos permite que podamos intuir algunos de los dibujos de instantes anteriores, con lo que podremos recrear un poco el efecto de estelas detrás de las pelotas, según se mueven. Prueba a variar este número para ver cómo resulta el efecto.

- Se crea una nueva instancia de la pelota Ball() usando un número aleatorio mediante la función **random()**. Entonces se añade este elemento al final del arreglo de las pelotas con el método push() hasta un máximo de 25 elementos. Podemos modificar dicho número pero teniendo en cuenta que un número muy elevado de pelotas podría ralentizar significativamente la animación.

- Se recorre el bucle por todo el conjunto de pelotas balls y se ejecuta el método para dibujar, **draw()**, cada una de las pelotas y actualizar sus datos con **update()**.

- Se ejecuta la función de nuevo mediante el método **requestAnimationFrame()** - cuando este método está continuamente ejecutándose y llama a la misma función esto ejecutará la función de animación un determinado número de veces por segundo para crear una animación fluida. Tal y como puedes ver, dicho método está dentro de la función loop, que resulta ser igualmente la función que invoca por lo que esta última función está funcionando de modo **recursivo**.

Por último, se añade una línea de código para ejecutar la función **loop()** y lanzar la animación que haga rebotar las pelotas en los bordes de la pantalla:

```
80  loop();
```

Paso 7: Añadiendo la detección de colisiones.

Ahora que ya tenemos la animación básica que permite generar las pelotas y hacer que las mismas se desplacen y reboten contra los bordes de la ventana del navegador, vamos a actualizar nuestra aplicación para que detecte las colisiones entre las pelotas.

El primer paso, será añadir el siguiente código a continuación de donde se definió el método update() (en código de Ball.prototype.update) y antes de que se genere el array con las pelotas:

```
Ball.prototype.collisionDetect = function () {
  for (let j = 0; j < balls.length; j++) {
    if (!(this === balls[j])) {
      let dx = this.x - balls[j].x;
      let dy = this.y - balls[j].y;
      let distance = Math.sqrt(dx * dx + dy * dy);

      if (distance < this.size + balls[j].size) {
        balls[j].color = this.color = `rgb(${random(0, 255)},${random(
          0,
          255,
        )},${random(0, 255)})`;
      }
    }
  }
}
```

Esta función realiza los siguiente:

- El bucle for permite recorrer todas las pelotas para comprobar si colisionan con otra.
- Dentro del bucle, usamos un if para comprobar si la pelota que estamos mirando en ese ciclo del bucle for es la pelota que estamos mirando. No queremos mirar si una pelota ha chocado consigo misma. Para esto miramos si la pelota actual (es decir la pelota que está invocando al método que resuelve la detección de colisiones) es la misma que la indicada por el bucle. Usamos un operador **!** para indicar una negación en la comparación, así que el código dentro de la condición solo se ejecuta si estamos mirando dos pelotas distintas.
- Luego, se calcula la distancia entre dos pelotas con un algoritmo común que en primer lugar obtiene la distancia entre la cada pelota (this.x y this.y y el resto de pelotas del array balls[j].x y balls[j].y). Esto se explica mejor en el enlace detección de colisión 2D.
- Finalmente, el código lo que hace es cambiar el color de las pelotas cuando detecta una colisión. Se pueden hacer cosas mucho más complicadas, como que las pelotas reboten una con la otra de forma realista. Para desarrollar esos efectos de simulación física, los desarrolladores tienden a usar librerías de física como PhysicsJS, etc.

Por último, este método se debe llamar dentro de la función **loop()** justo después del balls[i].update():

```
91  for (let i = 0; i < balls.length; i++) {
92    balls[i].draw();
93    balls[i].update();
94    balls[i].collisionDetect();
```

Finalmente, nuestra aplicación ya está completa y podemos ahora ejecutarla para comprobar que funciona correctamente.

Webgrafía

[1] “Ejercicio práctico de construcción de objetos”, *MDN Web Docs*. [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Object_building_practice [Consultado: 10-nov-2023].