FEW Labo: Javascript OO -Balls

1 Inhoud

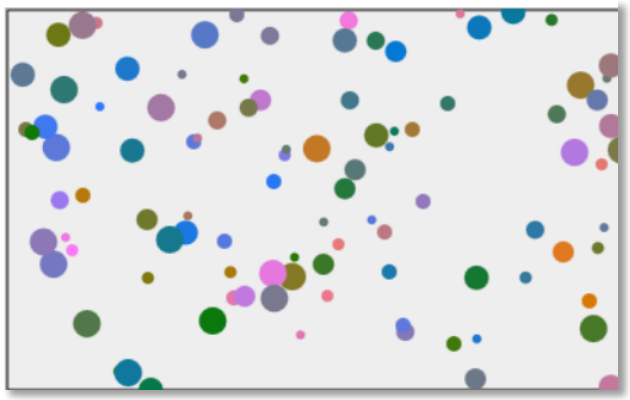
2 Doelstelling

- Object Oriented javascript inoefenen.
- Gebruik van constructor pattern en module pattern.
- Werken met “this” in constructor pattern.
- Aanbrengen van default waarden in een object met een verkorte if.
- Tekenen met javascript in de Canvas.
- Clickable event aanmaken in de Canvas.
- Gebruik van een closure om een animatie uit te voeren.

3 Balls in de canvas

3.1 Opgave

Een reeks ballen met verschillende grote en kleur bewegen aan verschillende snelheid binnen de canvas en botsen terug op de wand ervan.



3.2 Uitwerken van het canvas object. (Canvas.js)

1. Plaats een canvas in de html en test of de browser het canvas object ondersteunt.
```
<canvas id="canvas" width="400" height="250" style="border: 1px solid
#c3c3c3;">
    Uw browser ondersteunt jammer genoeg de canvas functie niet.
</canvas>
```

2. We wensen zo weinig mogelijk globale variabelen. De enige globale variabelen worden 3 javascript objecten.
```
<script src="scripts/Canvas.js"></script>
<script src="scripts/Ball.js"></script>
<script src="scripts/app.js"></script>
```

3. Uitwerking van het **Canvas object**.
   De basis setup van de Canvas komt ook in object om redenen van hergebruik.

   a. Het wordt opgebouwd volgens het constructor pattern waarbij alle nodige argumenten kunnen ingesteld worden.
   ```
   var Canvas = function (id, width, height, backgroundColor) {… }
   Canvas.prototype = { . . . .  }
   ```

   De eigenschappen in de constructor functie kan je al dan niet laten afhangen van aangebrachte eigenschappen in het html element. Daarbij wordt vaak ook het object zelf (hier de canvas) in een eigenschap ondergebracht in het prototype (of in de constructor functie).
   ```
   get obj() { return document.getElementById(this.id) ?
                      document.getElementById(this.id) : null }
   ```

   Nu het object beschikbaar is kunnen width, height al dan niet afhankelijk gemaakt worden van de html properties:
   ```
   this.width = width ? width : this.obj.width;
   ```

   Op eenzellfde manier kan je ook default waarden aanbrengen voor bvb. een achtergrondkleur. Je haalt hiervoor wel het style object op `this.obj.style`:

Test dit uit.

b. Voeg commentaar toe in het Canvas object volgens de JSdoc specificatie (http://usejsdoc.org/)

c. In Canvas.prototype worden verder de publiek beschikbare methoden aangebracht.

```
setContext: function () {
   if (this.obj && this.obj.getContext) {
       var oContext = this.obj.getContext('2d');

       oContext.fillStyle = this.backgroundColor;
       oContext.fillRect(0, 0, this.width, this.height);
       return (oContext ? oContext : null);
   }
```

Om deze context onmiddellijk uit te voeren, kon setContext() opgeroepen worden vanuit de constructor functie. We houden de returned context ook bij in een eigenschap: `this.ctx`. voor verder gebruik in de app.

```
this.ctx = this.setContext;
```

3.3 Eén clickable ball tekenen (Ball.js):

1. Uitwerking van **Ball.prototype.drawOneBall** -> tekenen van één bal.
   We maken gebruik van een object om verder meerdere ballen te kunnen genereren.
   We kiezen opnieuw voor het constructor pattern.
   a. Gezien we vectorieel werken binnen de canvas, zorgen we onmiddellijk voor verschillende bruikbare eigenschappen. Ook de canvas context geven we mee, omdat al het tekenen en hertekenen binnen een context van de canvas gebeurt.

```
function Ball(context, id, x, y, radius, speed, angle) {
    this.context = context;
    this.id = id;
    this.x = x ; // binnen de canvas - zonder + “px “!
    this.y = y;
    . . . . . ..

Ball.prototype = { }
```

   b. Het prototype zorgt voor methodes en eigenschappen over alle aangemaakte instanties. Er zijn in ieder geval een aantal gemeenschappelijke zaken nodig(= geldig voor alle instanties), die we daarom in dit prototype plaatsen.

```
get context() { return this.canvas.context },
get randomColor() {
        return  "rgb(" + Math.floor(Math.random() * 255) + ", " +
                         Math.floor(Math.random() * 255) + ", " +
                         Math.floor(Math.random() * 255) +
                 ")"
    }
```

   c. Om te beginnen tekenen we slechts één bal, in een willekeurige kleur op een willekeurige plaats. Binnen het prototype maken we een drawOneBall methode aan.
   Om te weten hoe je een cirkel tekent in de canvas, doe je beroep op referenties. Je hebt onder andere de context nodig, de posities, de straal (radius) …Zie http://www.html5canvastutorials.com/tutorials/html5-canvas-circles/

```
drawOneBall: function () {
```

```
            this.context.beginPath();
            this.context.arc( . . . . . .
            . . . . .
        }
```

2. Tijd om een eerste keer te testen in **app.js**
    a. In app.js laden we het DOM in met een EventListener.
    Daarin starten we de canvas applicatie op
    ```
    document.addEventListener("DOMContentLoaded", function () {
        //1. canvas initialisatie
        var oCanvas = new Canvas("canvas", null, null, "#eee");
        // 2. opstarten van de applicatie
        canvasApp(oCanvas);
    })
    ```

    b. In canvasApp testen we het tekenen van één bal.
    ```
    function canvasApp(oCanvas) {
            var oBall = new Ball(oCanvas, "ball1", . . . .). . . .
            oBall.drawOneBall();
    ```

3. Een clickable ball.
Bij het clicken van één bal, wensen we dat automatisch een tweede bal getekend wordt op een andere positie met een ander kleur.

    a. Click event toevoegen in de constructor.
    We krijgen een probleem met "this". Elke functie maakt zijn eigen scope. Dit betekent dat de innerfunctie van de eventHandler zijn eigen scope aanmaakt, waardoor je niet meer in direct contact staat met het nieuw aangemaakte "this"object.
    Mogelijke oplossingen:
    aanmaak van `var self = this` buiten de handler waarna je self gebruikt erbinnen:
    ```
    return self.onClick(e)
    ```

    of gebruik van de bind() methode.
    ```
    context.canvas.addEventListener('click',
            function (e) { return this.onClick(e) }.bind(this))
    ```

    b. De clickhandler uitwerken in prototype.
    Enkel op het canvas object kan een clickevent toegevoegd worden. Om te controleren of we wel degelijk op een bestaande ball instantie klikken, moet er bijkomende controle komen op de x en y posities.
    ```
    onClick: function (e) {
            if ((e.x > this.x - this.radius) && (e.x < this.x +
    this.radius)&&( //aanvullen voor y) {
                this.drawOneBall()
            }
        }
    ```

    c. Testen. Dit geeft niet het gewenste resultaat. Je krijgt geen fout, maar ook geen nieuwe bal. De oorzaak zit weer in "this": de zelfde instantie wordt hertekend. Een algemene oplossing hiervoor: geef via de constructor een nieuwe bal mee:
    ```
    this.drawOneBall(new Ball(this.context))
    ```

    En pas de bestaande prototype methode aan om deze nieuwe bal te verwerken.
    Om ook nog this toe te laten, kan je gebruik maken van de verkorte if:
    ```
    drawOneBall: function (ball) {
            ball = ball ? ball : this;
            ball.context.beginPath();
            . . .
    }
    ```

En nog werkt het niet. Er ontbreken vermoedelijk constructor waarden voor enkele argumenten zoals x , y ; radius. Ook dit lossen we op in de constructor met verkorte ifs.

```
this.y = y ? y : Math.floor(Math.random() * this.context.canvas.height);
```

3.4 Eén bal laten bewegen

1. Vooraf: het principe om te tekenen in de canvas:
   Om de bal te laten bewegen voegden we in de constructor van Ball.js een snelheidsvector toe. Deze snelheidsvector is gekenmerkt door zijn snelheid (speed) en zijn hoek in graden(angle).
   a. Hoeken in javascript moeten naar radialen omgezet worden. We voegen een methode toe in prototype:
      ```
      get radians() { return this.angle * Math.PI / 180 }
      ```
   b. De vector wordt vertaald naar een x en een y verplaatsing
      ```
      get vx() { return Math.cos(this.radians) * this.speed },
      get vy() { return  . ; . … .  },
      ```
   c. Animeren in de canvas is anders dan animeren in de browser. In de browser is het voldoende om een element te verplaatsen om de x msec met bvb. setTimeout(), setInterval() of Request Animation Frame. In de canvas moet alles steeds opnieuw getekend worden. Ook de context van de canvas! Onze bal verplaatsen betekent daarom:
      i. Het clearen van de context met:
         `oContext.clearRect(0, 0, canvasWidth, canvasHeight);`
         Of een alternatief: Het opnieuw opvullen van de canvas met zijn oorspronkelijke kleur en rand.
      ii. De nieuwe waarden van de animatie ophalen.
      iii. Het hertekenen van de context (elementen) met de nieuwe waarden.
      iv. Na een timeout of interval van x msec wordt het eerste puntje i herhaald.

2. Enkele tips:
   a. Clearen van de context = hertekenen van de rectangle:
      ```
      this.context.clearRect(0, 0, this.context.canvas.width,
      this.context.canvas.height)
      ```
   b. Tekenen doe je door `binnen een setTimeout` de eerder aangemaakte methode `drawOneBall()` op te roepen.
      Door het clearen van de context, lukt het niet meer om te klikken op meerdere ballen (kan je later oplossen).
   c. Bewegen is het toevoegen van vx en vy binnen setTimeout:
      ```
      this.x = this.x + this.vx;
      ```
   d. Test uit en los scope problemen op volgens eerder geziene techieken .

3. Voeg een methode " checkborder" toe aan het prototype. Deze methode zorgt ervoor dat
   a. De bal detecteert als een rand van de canvas bereikt werd:
      ```
      if (ball.x > ball.context.canvas.width || ball.x < 0) { }
      ```
   b. Waarna de hoek omgekeerd wordt en de richting veranderd wordt:
      ```
      ball.angle = 180 - ball.angle;
      ```

3.5 Meerdere ballen laten bewegen:

1. Principe:
   Om meerdere ballen te laten bewegen en toch telkens de canvas te clearen voor het hertekenen van alle ballen op hun nieuwe positie, wordt volgende techniek toegepast.
   a. Maak in app.js met een for lus meerdere ballen aan.
   b. Hou de status van deze ballen bij in een Array. De array is een shared array van het ball object: Ball.arrBalls.
   c. Bij elke teken cyclus van de canvas, worden alle ballen uit de array opgehaald en getekend op de canvas.

2. Constructor functie van Ball.js aanvullen met een Array:.
   ```
   Ball.arrBalls = typeof (Ball.arrBalls) == "undefined" ? [] : Ball.arrBalls;
   ```

3. Aanmaak in app.js van meerdere ballen.
   Met een `for` lus worden meerdere instanties van Ball.js aangemaakt.

   Wanneer default waarden zouden ontbreken (= niet aangemaakt in Ball.js of wanneer null) of wanneer je specifieke waarden wil aanbrengen maak je gebruik van javascript object om deze waarden aan te brengen. Voorbeeld
   ```
   var current = {
             speed: 2,
             angle: Math.floor(Math.random() * 360)
               }

   oBall = new Ball(oCanvas.ctx, "oBall", null, null, null, current.speed,
   current.angle);
   ```

4. Voeg de aangemaakte ballen toe aan de Array.
   ```
   Ball.arrBalls.push(oBall);
   ```

5. We willen tekenen met een methode: `oBall.draw()`. Deze methode zal de array uitlezen en elke bal tekenen.
   a. Maak de method draw aan in Ball.prototype:
   b. Als eerste stap in sdraw() wordt de context gecleared.
   c. Nadien wordt elke bal opgehaald uit de array met een `for` lus.
   d. Voor elke bal in de for lus wordt het volgende gedaan:
      i. Nieuwe waarden ophalen voor x en y met vx en vy
      ii. Tekenen van de ball: `this.drawOneBall(ball);`
      iii. Controleren of de canvas rand bereikt is: `this.checkBorder(ball)`

6. Test of er nu meerdere ballen getekend worden.

7. Om de ballen te laten bewegen maken we in app.js van een closure. In deze closure blijft de eerder aangemaakt oBall beschikbaar.
   ```
   function redraw() {
       oBall.draw()
   }
   ```

8. Om het even waar in een parent van de closure (= een outer functie) kan de animatie geactiveerd worden:
   ```
   setInterval(function () { redraw() }, 25);
   ```

3.6 De app opstarten met het module pattern.

De toepassing werkt wel en is af. Tijdens de oefening werd het constructor pattern gebruikt met bijgevolg veel aandacht voor het keywoord this. In het module pattern wordt this niet (veel minder) gebruikt. Het opstarten van de app (app.js) gaan we wijzigen naar een module pattern. Literatuur spreekt over het sneller opkuisen van de scope via het module pattern.

Bouw app.js om naar het module pattern:

```
var MyApp = (function() {
	function init() { //DOM inladen  }
	function startApp() {//canvas obj ophalen en app starten  }
	return {
		init: init,
		startApp: startApp
	};
})();

MyApp.init();
```
