> Cette page a été traduite à partir de l'anglais par la communauté. Vous pouvez également contribuer en rejoignant la communauté francophone sur MDN Web Docs.

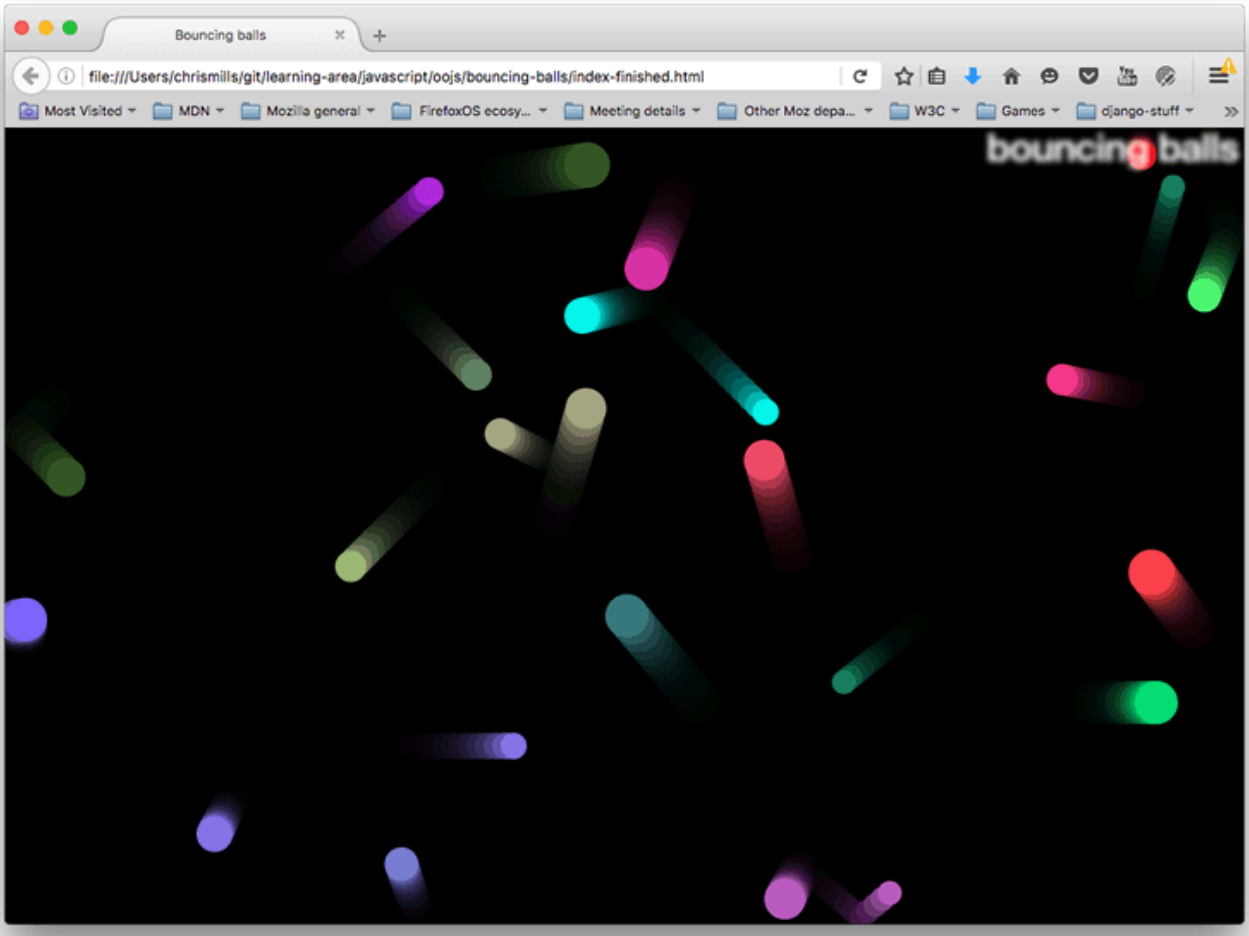
La construction d'objet en pratique

Dans l'article précédent, nous avons passé en revue l'essentiel de la théorie de l'objet Javascript et sa syntaxe détaillée, vous donnant ainsi des bases solides sur lesquelles commencer. Dans le présent article nous plongeons dans un exercice pratique afin d'accroître votre savoir-faire dans la construction d'objets entièrement personnalisés donnant un résultat plutôt amusant et très coloré.

Pré-requis :	Connaissance basique de l'informatique, une compréhension basique du HTML et du CSS, une familiarité avec les bases du JavaScript (voir Premiers pas et Les blocs de construction) et les bases de la programmation objet en JavaScript (voir Introduction aux objets (en-US)).
Objectif :	Acquérir plus de pratique dans l'utilisation des objets et des techniques orientées objet dans un contexte "monde réel".

Faisons bondir quelques balles

Dans cet article, nous écrirons une démo classique de "balles bondissantes", pour vous montrer à quel point les objets peuvent être utiles en JavaScript. Nos petites balles bondiront partout sur notre écran et changeront de couleurs lorsqu'elles se toucheront. L'exemple finalisé ressemblera un peu à ceci :



Cet exemple utilise l'API Canvas pour dessiner les balles sur l'écran, et l'API requestAnimationFrame pour animer l'ensemble de l'affichage — Nul besoin d'avoir une connaissance préalable de ces APIs, nous espérons qu'une fois cet article terminé, vous aurez envie d'en faire une exploration approfondie. Tout le long du parcours nous utiliserons certains objets formidables et vous montrerons nombre de techniques sympathiques comme des balles bondissantes sur les murs et la vérification de balles qui s'entrechoquent (encore connue sous l'appelation **détection de collision**).

Pour commencer, faites des copies locales de nos fichiers `index.html` , `style.css` , et `main.js` . Ces fichiers contiennent respectivement :

1. Un document HTML très simple contenant un élément `<h1>` , un élément `<canvas>` pour dessiner nos balles dessus et des élements pour appliquer notre CSS et notre JavaScript à notre HTML ;
2. Quelques styles très simples qui servent principalement à mettre en forme et placer le `<h1>` , et se débarasser de toutes barres de défilement ou de marges autour du

pourtour de notre page (afin que cela paraisse plus sympathique et élégant) ;

3. Un peu de JavaScript qui sert à paramétrer l'élément `<canvas>` et fournir les fonctions globales que nous utiliserons.

La première partie du script ressemble à ceci :

JS

```
const canvas = document.querySelector("canvas");

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

const width = (canvas.width = window.innerWidth);
const height = (canvas.height = window.innerHeight);
```

Ce script prend une référence à l'élément `<canvas>` et ensuite invoque la méthode `getContext()` sur lui, nous donnant ainsi un contexte sur lequel nous pouvons commencer à dessiner. La variable résultante (`ctx`) est l'objet qui représente directement la surface du Canvas où nous pouvons dessiner et qui nous permet de dessiner des formes 2D sur ce dernier.

Après, nous configurons les variables `width` (largeur) et `height` (hauteur), et la largeur et la hauteur de l'élément canvas (représentés par les propriétés `canvas.width` et `canvas.height`) afin qu'elles soient identiques à la fenêtre du navigateur (la surface sur laquelle apparaît la page web— Ceci peut être tiré des propriétés `Window.innerWidth` et `Window.innerHeight`).

Vous verrez qu'ici nous enchaînons les assignations de valeurs des différentes variables ensemble à des fins de rapidité. Ceci est parfaitement autorisé.

Le dernier morceau du script ressemble à ceci :

JS

```
function random(min, max) {
  var num = Math.floor(Math.random() * (max - min + 1)) + min;
  return num;
}
```

Cette fonction prend deux nombres comme arguments, et renvoie un nombre compris entre les deux.

Modéliser une balle dans notre programme

Notre programme met en œuvre beaucoup de balles bondissant partout sur l'écran. Comme nos balles se comporteront toutes de la même façon, cela semble tout à fait sensé de les représenter avec un objet. Commençons donc en ajoutant le constructeur suivant à la fin de notre code.

JS

```
function Ball(x, y, velX, velY, color, size) {
  this.x = x;
  this.y = y;
  this.velX = velX;
  this.velY = velY;
  this.color = color;
  this.size = size;
}
```

Ici, nous incluons des paramètres qui définissent des propriétés dont chaque balle aura besoin pour fonctionner dans notre programme :

- Les coordonnées `x` et `y` — les coordonnées verticales et horizontales où la balle débutera sur l'écran. Ceci peut se trouver entre 0 (coin à gauche en haut) et la valeur de la hauteur et de la largeur de la fenêtre du navigateur (coin en bas à droite).
- Une vitesse horizontale et verticale (`velX` et `velY`) — à chaque balle est attribuée une vitesse horizontale et verticale; en termes réels, ces valeurs seront régulièrement ajoutées aux valeurs de la coordonnée `x` / `y` quand nous commencerons à animer les balles, afin de les faire bouger d'autant sur chaque vignette (frame).
- Une couleur `color` — chaque balle a une couleur.
- Une taille `size` — chaque balle a une taille. Ce sera son rayon mesuré en pixels.

Dessiner la balle

En premier lieu, ajoutez la méthode `draw()` au `prototype` de `Ball()` :

JS

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

En utilisant cette fonction, nous pouvons dire à notre balle de se dessiner sur l'écran en appelant une série de membres du contexte 2D du canvas que nous avons défini plus tôt (`ctx`). Le contexte est comme le papier et maintenant nous allons demander à notre stylo d'y dessiner quelque chose :

- Premièrement, nous utilisons `beginPath()` pour spécifier que nous voulons dessiner une forme sur le papier.
- Ensuite, nous utilisons `fillStyle` pour définir de quelle couleur nous voulons que la forme soit — nous lui attribuons la valeur de la propriété `color` de notre balle.
- Après, nous utilisons la méthode `arc()` pour tracer une forme en arc sur le papier. Ses paramètres sont :
  - Les positions `x` et `y` du centre de l'arc — nous spécifions donc les propriétés `x` et `y` de notre balle.
  - Le rayon de l'arc — nous spécifions la propriété `size` de notre balle.
  - Les deux derniers paramètres spécifient l'intervalle de début et de fin en degrés pour dessiner l'arc. Ici nous avons spécifié 0 degré et `2 * PI` qui est l'équivalent de 360 degrés en radians (malheureusement, vous êtes obligés de spécifier ces valeurs en radians et non en degrés). Cela nous donne un cercle complet. Si vous aviez spécifié seulement `1 * PI` , vous auriez eu un demi-cercle (180 degrés).
- En dernière position, nous utilisons la méthode `fill()` qui est habituellement utilisée pour spécifier que nous souhaitons mettre fin au dessin que nous avons commencé avec `beginPath()` , et remplir la surface délimitée avec la couleur que nous avions spécifiée plus tôt avec `fillStyle` .

Vous pouvez déjà commencer à tester votre objet :

1. Sauvegardez le code et chargez le fichier html dans un navigateur.
2. Ouvrez la console JavaScript du navigateur et actualisez la page afin que la taille du canvas change et prenne la petite taille restante de la fenêtre lorsque la console est ouverte.
3. Tapez dans la console ce qui suit afin de créer une nouvelle instance de balle :

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

4. Essayez d'appeler ses membres :

```
JS
testBall.x;
testBall.size;
testBall.color;
testBall.draw();
```

5. Lorsque vous entrerez la dernière ligne, vous devriez voir la balle se dessiner quelque part sur votre canvas.

Mettre à jour les données de la balle

Nous pouvons dessiner la balle dans n'importe quelle position, mais actuellement pour commencer à la bouger, nous aurons besoin d'une sorte de fonction de mise à jour. Insérez donc le code suivant à la fin de votre fichier JavaScript pour ajouter une méthode `update()` au `prototype` de `Ball()` :

```
JS
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;
};
```

Les quatre premières parties de la fonction vérifient si la balle a atteint le rebord du `canvas`. Si c'est le cas, nous inversons la polarité de la vitesse appropriée pour faire bouger la balle dans le sens opposé. Donc, par exemple, si la balle se déplaçait vers le haut (`velY` positif) alors la vitesse verticale est changée afin qu'elle commence à bouger plutôt vers le bas (`velY` négatif).

Dans les quatre cas, nous vérifions :

- Si la coordonnée `x` est plus grande que la largeur du `canvas` (la balle est en train de sortir du côté droit).
- Si la coordonnée `x` est plus petite que `0` (la balle est en train de sortir du côté gauche).
- Si la coordonnée `y` est plus grande que la hauteur du `canvas` (la balle est en train de sortir par le bas).
- Si la coordonnée `y` est plus petite que `0` (la balle est en train de sortir par le haut).

Dans chaque cas, nous incluons la taille (`size`) de la balle dans les calculs parce que les coordonnées `x` / `y` sont situées au centre de la balle, mais nous voulons que le pourtour de la balle rebondisse sur le rebord — nous ne voulons pas que la balle sorte à moitié hors de l'écran avant de commencer à rebondir vers l'arrière.

Les deux dernières lignes ajoutent la valeur `velX` à la coordonnée `x` et la valeur `velY` à la coordonnée `y` — la balle est en effet mise en mouvement chaque fois que cette méthode est invoquée.

Cela suffira pour l'instant, passons à l'animation !

Animer la balle

Maintenant, rendons cela amusant. Nous allons commencer à ajouter des balles au canvas et à les animer.

1. Tout d'abord, nous avons besoin d'un endroit où stocker toutes nos balles. Le tableau suivant fera ce travail — ajoutez-le au bas de votre code maintenant :

   JS

```
let balls = [];

while (balls.length < 25) {
  let size = random(10, 20);
  let ball = new Ball(
    // ball position always drawn at least one ball width
    // away from the edge of the canvas, to avoid drawing errors
    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);
}
```

   Tous les programmes qui animent les choses impliquent généralement une boucle d'animation, qui sert à mettre à jour les informations dans le programme et à restituer ensuite la vue résultante sur chaque image de l'animation. C'est la base de la plupart des jeux et autres programmes similaires.

2. Ajoutez ce qui suit au bas de votre code maintenant :

   JS

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

  for (let i = 0; i < balls.length; i++) {
```

```
      balls[i].draw();
      balls[i].update();
    }

    requestAnimationFrame(loop);
  }
```

Notre fonction `loop()` fonctionne comme suit :

- On définit la couleur de remplissage du canvas en noir semi-transparent, puis dessine un rectangle de couleur sur toute la largeur et la hauteur du canvas, en utilisant `fillRect()` (les quatre paramètres fournissent une coordonnée de départ, une largeur et une hauteur pour le rectangle dessiné). Cela sert à masquer le dessin de l'image précédente avant que la suivante ne soit dessinée. Si vous ne faites pas cela, vous verrez juste de longs serpents se faufiler autour de la toile au lieu de balles qui bougent ! La couleur du remplissage est définie sur semi-transparent, `rgba (0,0,0,.25)`, pour permettre aux quelques images précédentes de briller légèrement, produisant les petites traînées derrière les balles lorsqu'elles se déplacent. Si vous avez changé 0.25 à 1, vous ne les verrez plus du tout. Essayez de faire varier ce dernier nombre (entre 0 et 1) pour voir l'effet qu'il a.
- On crée un nouvel objet `Ball()` avec des attributs générés aléatoirement grâce à la fonction `random()`, puis on ajoute l'objet au tableau, mais seulement lorsque le nombre de balles dans le tableau est inférieur à 25. Donc quand on a 25 balles à l'écran, plus aucune balle supplémentaire n'apparaît. Vous pouvez essayer de faire varier le nombre dans `balls.length <25` pour obtenir plus, ou moins de balles à l'écran. En fonction de la puissance de traitement de votre ordinateur / navigateur, spécifier plusieurs milliers de boules peut ralentir l'animation de façon très significative !
- Le programme boucle à travers tous les objets du tableau sur chacun desquels il exécute la fonction `draw()` et `update()` pour dessiner à l'écran chaque balle et faire les mises à jour de chaque attribut avant le prochain rafraîchissement.
- Exécute à nouveau la fonction à l'aide de la méthode `requestAnimationFrame()` — lorsque cette méthode est exécutée en permanence et a reçu le même nom de fonction, elle exécute cette fonction un nombre défini de fois par seconde pour créer une animation fluide. Cela se fait généralement de manière récursive — ce qui signifie que la fonction s'appelle elle-même à chaque fois qu'elle s'exécute, de sorte qu'elle sera répétée encore et encore.

3. Finallement mais non moins important, ajoutez la ligne suivante au bas de votre code — nous devons appeler la fonction une fois pour démarrer l'animation.

JS

```
loop();
```

Voilà pour les bases — essayez d'enregistrer et de rafraîchir pour tester vos balles bondissantes!

Ajouter la détection de collision

Maintenant, pour un peu de plaisir, ajoutons une détection de collision à notre programme, afin que nos balles sachent quand elles ont frappé une autre balle.

1. Tout d'abord, ajoutez la définition de la méthode suivante sous la définition de la méthode `update()` (c'est-à-dire le bloc `Ball.prototype.update`) :

JS

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

Cette méthode est un peu complexe, donc ne vous inquiétez pas si vous ne comprenez pas exactement comment cela fonctionne pour le moment. Regardons cela pas-à-pas :

- Pour chaque balle *b*, nous devons vérifier chaque autre balle pour voir si elle est entrée en collision avec *b*. Pour ce faire, on inspecte toutes les balles du tableau `balls[]` dans une boucle `for` .
- Immédiatement à l'intérieur de cette boucle `for` , une instruction `if` vérifie si la balle courante *b'* , inspectée dans la boucle, n'est pas égale à la balle *b*. *Le code correspondant est :* `b'!== b` . En effet, nous ne voulons pas vérifier si une balle *b* est entrée en collision avec elle-même ! Nous contrôlons donc si la balle actuelle *b* —dont la méthode `collisionDetect()` est invoquée—est distincte de la balle *b'* inspectée dans la boucle*.* Ainsi le bloc de code venant après l'instruction `if` ne s'exécutera que si les balles *b* et *b'* ne sont pas identiques.
- Un algorithme classique permet ensuite de vérifier la superposition de deux disques. Ceci est expliqué plus loin dans 2D collision detection.
- Si une collision est détectée, le code à l'intérieur de l'instruction interne `if` est exécuté. Dans ce cas, nous définissons simplement la propriété `color` des deux cercles à une nouvelle couleur aléatoire. Nous aurions pu faire quelque chose de bien plus complexe, comme faire rebondir les balles de façon réaliste, mais cela aurait été beaucoup plus complexe à mettre en œuvre. Pour de telles simulations de physique, les développeurs ont tendance à utiliser des bibliothèques de jeux ou de physiques telles que PhysicsJS , matter.js , Phaser , etc.

2. Vous devez également appeler cette méthode dans chaque image de l'animation. Ajouter le code ci-dessous juste après la ligne `balls[i].update();` :

   JS

   ```
   balls[i].collisionDetect();
   ```

3. Enregistrez et rafraîchissez la démo à nouveau, et vous verrez vos balles changer de couleur quand elles entrent en collision !

> **Note :** Si vous avez des difficultés à faire fonctionner cet exemple, essayez de comparer votre code JavaScript avec notre version finale (voir également la démo en ligne).

Résumé

Nous espérons que vous vous êtes amusé à écrire votre propre exemple de balles aléatoires bondissantes comme dans le monde réel, en utilisant diverses techniques orientées objet et divers objets d'un bout à l'autre du module ! Nous espérons vous avoir offert un aperçu utile de l'utilisation des objets.

C'est tout pour les articles sur les objets — il ne vous reste plus qu'à tester vos compétences dans l'évaluation sur les objets.

Voir aussi

- Didacticiel sur canvas — un guide pour débutants sur l'utilisation de canvas 2D.
- requestAnimationFrame()
- Détection de collision 2D
- Détection de collision 3D (en-US)
- Jeu d'évasion 2D utilisant du JavaScript pu —un excellent tutoriel pour débutant montrant comment construire un jeu en 2D.
- Jeu d'évasion 2D utilisant phaser — explique les bases de la construction d'un jeu 2D en utilisant une bibliothèque de jeux JavaScript.

This page was last modified on 3 août 2023 by MDN contributors.