Jeu du casse-briques en 2d en javascript

Nous allons créer pas à pas un casse-briques entièrement en javascript et sans framework. En utilisant la balise HTML5 **<canvas>**.

Etapes 1 La page HTML du jeu :

Structure html très simple, tout le jeu sera contenu dans l'élément **<canvas>** , cet élément aura 3 attribut : un id , une longueur **width=480px** et une hauteur **height=320px**

Voici le code :

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Casse Briques</title>
    <style>
     * { padding: 0; margin: 0; }
     canvas { background: #060606; display: block; margin: 0 auto; }
    </style>
</head>
<body>

    <canvas id="monCanvas" width="480" height="320"></canvas>

    <script>

        //Votre code javascript ici

    </script>

</body>
</html>
```

Les bases de canvas

Pour utiliser l'élément **<canvas>,** pour le rendu graphique, il faut d'abord en donner la référence à Javascript. Ajouter ce code après la balise **<script>**

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

On enregistre d'abord la référence à l'élément **<canvas>** dans une variable nommée **canvas**, puis on crée la variable **ctx** pour stocker le contexte 2d, l'outil qui va nous permettre de dessiner sur canvas.

Voici un exemple de code qui dessine un carré rouge :

```
ctx.beginPath();
ctx.rect(20, 40, 50, 50);
ctx.fillStyle = "#FF0000";
ctx.fill();
ctx.closePath();
```

Toutes les instructions se situe entre **beginPath()** et **closePath()** . On définit un rectangle en utilisant **rect()** : les 2 premières valeurs spécifient les coordonnées du coin supérieurs gauche du rectangle, les 2 autres spécifient la largeur et la hauteur. Dans l'exemple le rectangle est dessiné à 20 pixels du coté gauche de l'écran et à 40 pixels du haut, et à une largeur de 50 pixels et une hauteur de 50 pixels. La propriété fillStyle stocke une couleur qui sera utilisée par la methode **fill()** pour dessiné le carré en rouge.

Voici un autre exemple pour dessiner un cercle vert :

```
ctx.beginPath();
ctx.arc(240, 160, 20, 0, Math.PI*2, false);
ctx.fillStyle = "green";
ctx.fill();
ctx.closePath();
```

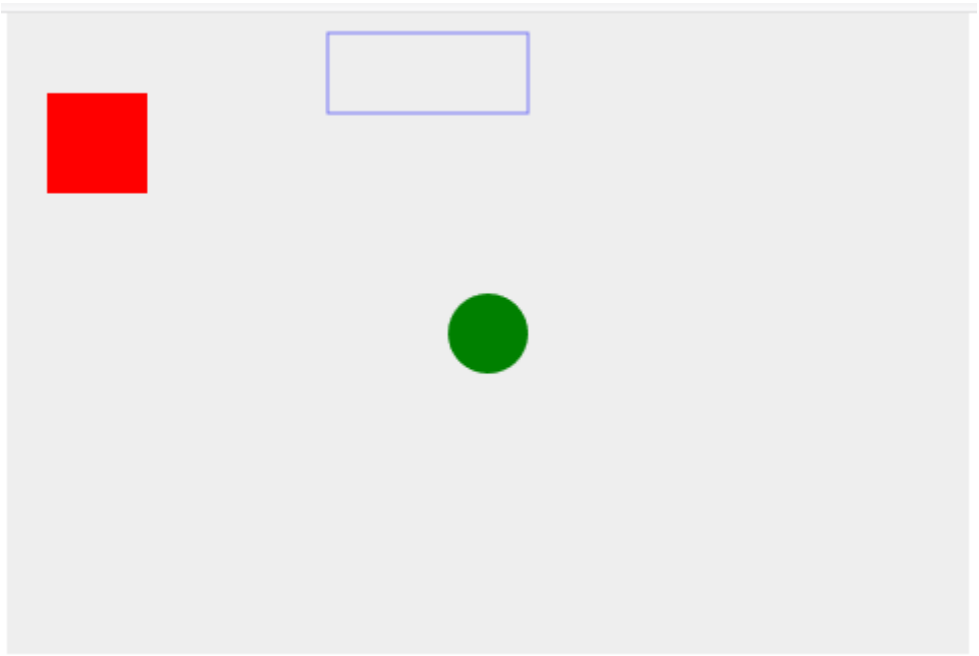
Cette fois ci on utilise la méthode **arc()** qui prend 6 paramètres :

- Les coordonnées **x** et **y** du centre du cercle
- Rayon du cercle
- L'angle de départ et l'angle de fin(pour finir de dessiner le cercle, en radiant)
- Direction du dessin(**false** pour le sens des aiguilles d'une montre(par défaut) ou **true** pour le sens inverse). Ce dernier paramètre est facultatif.

Au lieu d'utiliser **fillStyle** et dessiner des formes pleines, on peut aussi utiliser **stroke()** pour ne colorer que le contour extérieur.

Exemples :

```
ctx.beginPath();
ctx.rect(160, 10, 100, 40);
ctx.strokeStyle = "rgba(0, 0, 255, 0.5)";
ctx.stroke();
ctx.closePath();
```



Etapes 2 Définir une boucle pour le dessin

Le dessin doit être constamment mis à jour sur le canevas à chaque image, pour ce faire on va définir une fonction **draw()** qui sera executée en continu. En javascript pour qu'une fonction s'execute de façon repetée on peut utiliser les méthodes **setInterval()** ou **requestAnimationFrame()**.

Voici le code de départ de la fonction **draw()** qui sera executé toutes les 10 millisecondes grâce à **setInterval()** :

```
function draw() {
  // le code pour dessiner
}
setInterval(draw, 10);
```

Grâce à la nature infinie de **setInterval()**, la fonction **draw()** sera appelée toutes les 10 millisecondes jusqu'à ce qu'on l'arrêtes.

Voici le code à ajouter dans la fonction **draw()** pour dessiner la balle :

```
ctx.beginPath();
ctx.arc(50, 50, 10, 0, Math.PI*2);
ctx.fillStyle = "#ed151b";
ctx.fill();
ctx.closePath();
```

Etapes 3 Déplacer la balle

Pour l'instant on ne voie pas l'image se redessiner car elle reste immobile, pour la faire bouger on va remplacer la position bloquée à (50,50) par un point de départ en bas et au milieu du canevas grâce aux variable **posX** et **posY** que nous utiliserons pour définir la position où le cercle est dessiné

Ajouter ces 2 variables pour définir **posX** et **posY** :

```
var posX = canvas.width/2;
var posY = canvas.height-30;
```

Puis il faut remplacer les coordonnées du centre du cercles par les 2 variables comme ceci :

```
function draw() {

    ctx.beginPath();
    ctx.arc(posX, posY, 10, 0, Math.PI*2);
    ctx.fillStyle = "#ed151b";
    ctx.fill();
    ctx.closePath();

}
```

Ensuite on va ajouter une valeur à **posX** et **posY** après que chaque images est été dessiner pour donner un effet de déplacement, on définit 2 variables **dx** et **dy** avec comme valeurs 2 et -2 :

```
var dx = 2;
var dy = -2;
```

Puis à chaque image on mettra à jour **posX** et **posY** avec les variables **dx** et **dy :**

```
function draw() {

    ctx.beginPath();
    ctx.arc(posX, posY, 10, 0, Math.PI*2);
    ctx.fillStyle = "#ed151b";
    ctx.fill();
    ctx.closePath();
    posX += dx;
    posY += dy;
}
```

Si on teste le code on a bien notre balle qui se déplace mais elle laisse une trainé derrière elle :

Effacer le canevas avant chaque image(frame)

La balle laisse une trace car un nouveau cercle est dessiné sans qu'on efface le précédent. Pour résoudre ce probléme on va utiliser la méthode **clearRect(),** elle prend 4 paramétres :

- Les coordonnées x et y du coin supérieur gauche d'un rectangle
- Les coordonnées x et y du coin inférieur droit d'un rectangle

Toute la zone couverte par ce rectangle sera effacée.

Voici la fonction **draw()** avec le code qui va effacé le canevas avant de dessiner une nouvelle image :

```
function draw() {

    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.beginPath();
    ctx.arc(posX, posY, 10, 0, Math.PI*2);
    ctx.fillStyle = "#ed151b";
    ctx.fill();
    ctx.closePath();
    posX += dx;
    posY += dy;
}
```

Toutes les 10 millisecondes, le canvas est effacé, la balle est dessinée sur une position donnée et les valeurs **posX** et **posY** sont mises à jour pour l'image suivante.

Nettoyer le code :

Par la suite on va ajouter de plus en plus d'instruction à la fonction **draw().** On va donc la garder le plus propres possible. On va donc déplacer le code qui dessine la balle dans une fonction séparée et qui sera appelée dans la fonction **draw()** :

```
function drawBall()
{
    ctx.beginPath();
    ctx.arc(posX, posY, 10, 0, Math.PI*2);
    ctx.fillStyle = "#ed151b";
    ctx.fill();
    ctx.closePath();
}

function draw() {

    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawBall();
    posX += dx;
    posY += dy;
}
```

Etapes 4 Détection des murs :

Pour l'instant notre balle se déplace mais ne rebondit pas, pour résoudre ce problème nous allons ajouter à notre code de quoi détecter les collisions avec les murs.

Pour détecter les collisions, il va falloir vérifier si la balle touche le mur, si elle rentre en collision, alors on change sa direction.

Pour simplifier les calculs on va d'abord créer une variable **ballRadius** qui contient le rayon de la balle :

```
var ballRadius = 10;
```

Puis on modifie le rayon dans la fonction **drawBall() :**

```
ctx.arc(posX, posY, ballRadius, 0, Math.PI*2);
```

Rebondir en haut et en bas :

A chaque rafraichissement il faut regarder si la balle touche le bord du haut, si c'est le cas alors on inverse la direction de la balle. Rappel : les coordonnées commencent du coin en haut à gauche.

On peut donc écrire :

```
if(posY + dy < 0 )
{
    dy = -dy;
}
```

Si la position en y de la balle est inférieur à 0, on change la direction du mouvement sur l'axe y en inversant le signe du déplacement de la balle.

Exemple : si elle bouge vers le haut (donc une vitesse de -2 car le point (0,0) est en haut à gauche) et qu'elle rencontre le mur du haut, alors sa vitesse passe à 2 et la balle va redescendre.

Pour le mur du bas le code est presque le même :

```
if(posY + dy > canvas.height)
{
    dy = -dy;
}
```

Si la position en y de la balle est supérieur à la hauteur du canvas (soit 480 pixels) on inverse encore comme précédemment la vitesse de la balle.

On peut rassembler les deux conditions en une grâce au "ou" qui s'écrit || en JavaScript :

```
if(posY + dy > canvas.height || posY + dy < 0 )
{
    dy = -dy;
}
```

Rebondir à gauche et à droite :

Pour les murs gauche et droite le principe est le même sauf que l'on va tester sur l'axe x et sur la largeur du canvas :

```
if(posX + dx > canvas.width || posX + dx < 0 )
{
    dx = -dx;
}
```

On peut donc intégrer ce code dans la fonction **draw()** juste après avoir dessiné la balle :

```
function draw() {

    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawBall();

    if(posX + dx > canvas.width || posX + dx < 0 )
    {
    dx = -dx;
    }


    if(posY + dy > canvas.height || posY + dy < 0 )
    {
    dy = -dy;
    }

    posX += dx;
    posY += dy;
}
```

La balle rebondit bien, mais cependant elle s'enfonce dans le mur avant de changer de direction, elle s'enfonce de la profondeur de son rayon.

Cela s'explique car on calcule la collision à partir des coordonnées x et y du centre de la balle, on ne prend pas en compte le rayon de la balle, pour résoudre ce problème on va ajouter le rayon de la balle à nos conditions comme ceci :

```
if(posX + dx > canvas.width-ballRadius || posX + dx < ballRadius )
{
dx = -dx;
}


if(posY + dy > canvas.height-ballRadius || posY + dy < ballRadius )
{
dy = -dy;
}
```

Maintenant quand la distance entre le centre de la balle et le bord est le même que le rayon de la balle cela change sa direction

Etapes 5 Création d'un paddle pour taper la balle

On va créer un rectangle qui sera notre paddle, pour cela on va définir 3 variables : une pour la hauteur, une pour la longueur et une pour son point de départ sur l'axe x :

```
var paddleHeight = 10;
var paddleWidth = 75;
var paddleX = (canvas.width-paddleWidth)/2;
```

Puis on crée une fonction **drawPaddle()** pour le dessiner :

```
function drawPaddle()
 {
    ctx.beginPath();
    ctx.rect(paddleX, canvas.height-paddleHeight, paddleWidth, paddleHeight);
    ctx.fillStyle = "#ed151b";
    ctx.fill();
    ctx.closePath();
}
```

Contrôle du paddle avec le clavier :

Pour ajouter des commandes claviers on aura besoins de :

- 2 variables pour déterminer si c'est le bouton gauche ou droite qui est pressé.
- 2 « event listeners » (des événements qui écoutes) : **keydown** et **keyup**
- 2 fonctions pour gérer **keydown** et **keyup**
- La possibilité de bouger à gauche et à droite

Pour les 2 variables on va utiliser un boolean initialisé à false :

```
var rightPressed = false;
var leftPressed = false;
```

Pour savoir si des touches sont pressées on va utiliser des « listeners », ajouter ces lignes juste avant le **setInterval() :**

```
document.addEventListener("keydown", keyDownHandler, false);
document.addEventListener("keyup", keyUpHandler, false);
```

Lorsque l'événement **keydown** est déclenché par l'appui d'une des touches de votre clavier (lorsqu'elles sont enfoncées), la fonction **keyDownHandler()** est exécutée. Le même schéma est vrai pour le deuxième **listener** : les événements

keyup activent la fonction **keyUpHandler()** (lorsque les touches cessent d'être enfoncées). Ajoutez ces lignes à votre code, sous les lignes **addEventListener()** :

```
function keyDownHandler(e)
 {

    if(e.keyCode == 39)
    {
     rightPressed = true;
    }
    else if(e.keyCode == 37)
     {
        leftPressed = true;
    }
}

function keyUpHandler(e)
{
    if(e.keyCode == 39)
    {
    rightPressed = false;
    }
    else if(e.keyCode == 37)
    {
    leftPressed = false;
    }
}

setInterval(draw, 10);
```

Quand on presse une touche du clavier, l'information est gardée dans une variable. La variable concernée est mise sur **true**. Quand la touche est relâchée, la variable revient à false.

Les deux fonctions prennent un paramètre, représenté par la variable e. Avec cette variable vous avez une information précieuse : le **keyCode** qui contient l'information indiquant quelle est la touche qui est pressée (vous pouvez aller faire un tour ici). Par exemple 37 est le **keycode** de la flèche gauche du clavier et 39 celui de la flèche droite. Si la gauche est pressée **leftPressed** est mis à **true**, et quand celle-ci est relâchée **leftPressed** est mis à false. Le même processus pour la flèche droite s'occupe de gérer **rightPressed**.

Fin de la première partie.

Exercice : faites en sorte de pouvoir déplacer le paddle de gauche à droite, sans que celui-ci sorte du canevas, essayer de gérer le problème avec un **if..else if** pour que le paddle ne sorte plus du canevas, n'oubliez pas de prendre en compte la taille du paddle dans le calcul des coordonnées.