1. L'ELEMENT <CANVAS> HTML5 – LE JEU DU CASSE BRIQUE PAS A PAS

L'objectif de ce TP est de découvrir l'élément **Canvas** de **HTML5**. Il permet de définir une zone où l'on peut créer des animations, des graphiques et des dessins … Cet élément se prête particulièrement bien au développement de jeux en 2D. Le langage **JavaScript** permet de piloter les éléments graphiques de cette zone. C'est donc une bonne occasion supplémentaire de renforcer nos connaissances en **JavaScript**. Pour cela, nous allons développer pas à pas le jeu du **Casse Brique** en **2D**.

Dans un premier temps nous allons définir notre espace de jeu, il suffit pour cela de créer une zone **Canvas** comme il suit :

index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Casse Brique</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <link rel="stylesheet" type="text/css" href="css/style.css" />
    <link rel="icon" type="image/png" href="img/casse_brique1.png">
    <!--[if lt IE 9]><script type="text/javascript"
src="js/excanvas.compiled.js"></script><![endif]-->
</head>
<body>
    <h1 id="montitre" class="colorElement">Casse Brique en JavaScript</h1>
    <br><br>
    <div id="contenu">
        <canvas id="myCanvas" width="480" height="320"></canvas>
        <input type="button" id="monbouton" value="Jouer">
        <label class="colorElement"
for="couleurback">     Couleur du Fond :</label>
        <input type="color" id="couleurback" value="#8d8787">
            
        <label class="colorElement" for="couleurtitre">  Couleur des
Eléments :</label>
        <input type="color" id="couleurtitre" value="#F50EB6">
    </div>
      <script>
            // Code JavaScript pour implémenter les divers fonctionnalités
            // du Jeu et de l'interface utilisateur.
      </script>
</body>
</html>
```

Sous le **Canvas**, nous définissons 3 Boutons, un bouton *Jouer* pour lancer le jeu, un bouton de type *color* pour changer la **couleur de fond** du **Canvas**, un dernier bouton de type *color* pour changer la **couleur des éléments** de l'interface utilisateur (libellés, titre, bale, briques, raquette).
Dans le script **CSS** (style.css) nous définissons de façon sommaire l'apparence par défaut des quelques éléments de l'interface utilisateur.

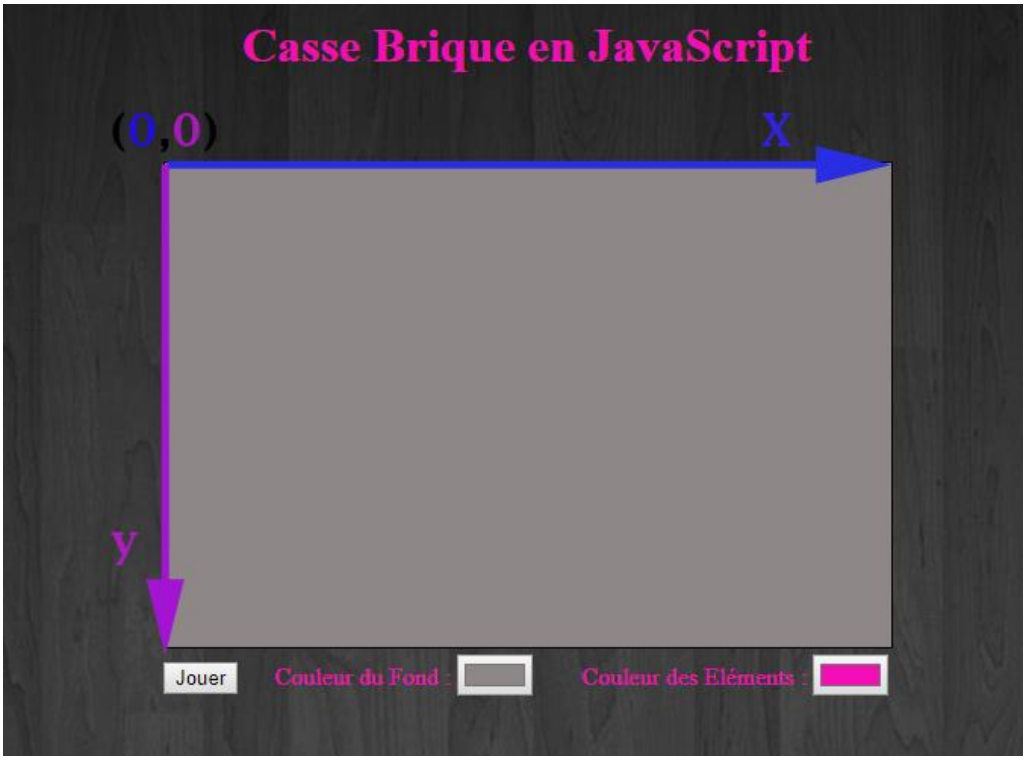
style.css

```css
body {
    background-image: url(../img/bg.jpg);
    color: #F50EB6;
}

canvas {
    border: 1px solid black;
    background-color: #8d8787;
}

h1 {
    text-align: center;
    color: #F50EB6;
}

#contenu {
    width: 480px;
    margin: auto;
}
```

Nous allons maintenant dessiner dans le **Canvas** notre premier élément graphique. Pour cela, nous disposons d'une API **JavaScript** spécifique à l'élément **Canvas**.

Dans un premier temps, nous devons donner la référence à **JavaScript** de notre **Canvas,** puis nous créons la variable **ctx** pour stocker le **contexte de rendu 2D :**

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



Cette capture d'écran vous permettra de bien comprendre le positionnement d'un élément graphique dans le **Canvas** en fonction de ses **coordonnées (X, Y)**.

Puis, nous allons dessiner notre premier élément graphique, la **balle** du jeu :

```
ctx.beginPath();
ctx.arc(200, 200, 10, 0, Math.PI * 2);
ctx.fillStyle = "#F50EB6";
ctx.fill();
ctx.closePath();
```

Toutes les méthodes de dessin doivent se trouver entre les méthodes **beginPath ()** et **closePath ().**
La propriété **fillStyle** stocke la **couleur** qui sera utilisée par la méthode **fill()** pour dessiner la **balle** en utilisant la méthode **arc()** à laquelle on passe les paramètres qui suivent :

- les coordonnées **x=200** et **y=200** sont les coordonnées du centre de l'**arc de cercle** que l'on veut tracer. (Ici c'est la **balle**).

- **10** représente le **rayon** de la **balle**.

- **0, Math.PI*2** représentent respectivement l'**angle de départ** et l'**angle de fin** en **radiant** pour dessiner la balle.

- **false** représente la **direction du dessin** de l'angle, ici **false** par défaut, pour le **sens des aiguilles d'une montre**, **true** pour le sens inverse.

Nous allons voir maintenant comment **animer** cette balle. Pour obtenir un effet de mouvement de la balle, nous devrons dessiner la **balle** dans une position légèrement différente et nous devrons effacer la balle dans sa position précédente. Nous sommes dans les principes utilisés pour créer des dessins-animés.

Pour garder constamment à jour le dessin du **Canvas** sur chaque image, nous devons définir une fonction **draw ()** exécutée en **continue**, avec un ensemble diffèrent de valeurs variables à chaque fois pour changer la position de la balle. Pour cela, nous pourrons utiliser les fonctions de **synchronisation JavaScript** : ***setInterval()*** ou ***requestAnimationFrame()***

La fonction **draw()** contiendra les méthodes de dessin vues précédemment. Pour obtenir une animation de la balle, nous devons définir des **coordonnées (x,y)** de son centre pour la position initiale de la balle :

```
var x = canvas.width/2;
var y = canvas.height-30;
```

Puis, nous devons modifier ces coordonnées à chaque image en opérant un **décalage** pour chacune de ces coordonnées soient **dx** et **dy** :

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

Ensuite, il suffira de mettre à jour les variables **x** et **y** pour chaque image. Ce que nous ferons dans la fonction **draw() :**

```
function draw() {
   ctx.beginPath();
   ctx.arc(x, y,10, 0, Math.PI * 2);
   ctx.fillStyle = "#F50EB6";
   ctx.fill();
   ctx.closePath();
   x += dx;
   y += dy;
}
setInterval(draw, 10);
```

Et nous appellerons cette fonction **draw()** toutes les 10 millisecondes grâce à la méthode **JavaScript setInterval(draw, 10)**.
Vous pouvez essayer le code implémenté jusqu'ici, vous remarquerez que la balle se met en mouvement mais qu'elle est suivie par une *trace résiduelle*. En effet, une **nouvelle balle** est **redessinée** toutes les **10 ms** mais sans effacer la précédente, pour avoir une impression de mouvement il faut effacer à chaque fois le contenu du **Canvas**.

Pour cela il existe une méthode spécifique : **clearRect()**

```
function draw() {
  ctx.beginPath();
```

```
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.arc(x, y,10, 0, Math.PI * 2);
  ctx.fillStyle = "#F50EB6";
  ctx.fill();
  ctx.closePath();
  x += dx;
  y += dy;
}
setInterval(draw, 10);
```

Cette méthode prend en compte quatre paramètres: les coordonnées **x** et **y** du coin **supérieur gauche** d'un rectangle et les coordonnées **x** et **y** du **coin inférieur droit** d'un rectangle. Toute la zone couverte par ce rectangle effacera tout le contenu dessiné précédemment.
Rafraichissez la fenêtre de votre navigateur après modification de votre code et vous obtiendrez bien une balle en mouvement ;-)

Afin d'améliorer la présentation de notre code, nous allons le scinder en deux fonctions distinctes :

```
function drawBall() {
   ctx.beginPath();
   ctx.arc(x, y, 10, 0, Math.PI * 2);
   ctx.fillStyle = "#F50EB6";
   ctx.fill();
   ctx.closePath();
}

function draw() {
   ctx.clearRect(0, 0, canvas.width, canvas.height);
   drawBall();
   x += dx;
   y += dy;
}
setInterval(draw, 10);
```

Jusqu'ici, nous avons **dessiné** notre **balle** et nous l'avons faite **bouger**, mais elle ne cesse de disparaître du bord de notre **Canvas**. Nous allons maintenant voir comment faire **rebondir la balle contre les bords du Canvas.**

Pour détecter la **collision**, nous allons vérifier si la *balle* est en contact avec le *mur* et, le cas échéant, nous allons *changer la direction de son mouvement* en conséquence.

Pour effectuer les calculs nous allons définir une variable *rayon de la balle* : **ballRadius**

```
var x = canvas.width / 2;
var y = canvas.height - 30;
var dx = 2;
var dy = -2;
var ballRadius = 10;
```

```
function drawBall() {
  ctx.beginPath();
  ctx.arc(x, y, ballRadius, 0, Math.PI * 2);
  ctx.fillStyle = "#F50EB6";
  ctx.fill();
  ctx.closePath();
}
```

Implémentons la détection de la **collision** de la **balle** avec le *bord supérieur* du ***Canvas.*** Si la balle touche le bord supérieur du **Canvas** et compte tenu de **l'orientation** des **axes de coordonnées** que nous avons vu précédemment**,** nous pouvons traduire l'effet de changement d'orientation après rebond par :

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

De la même façon nous pouvons raisonner pour le rebond sur le *bord inférieur* du **Canvas.**

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

Que nous pouvons résumer en une seule condition :

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

Sur le même principe, à vous de gérer les **collisions** sur les bords **Gauches** et **Droits** du **Canvas**.

Maintenant, vous remarquerez que la **balle** « *s'enfonce dans le mur* » jusqu'à son centre, effectivement nous avons géré les collisions par rapport au centre de la balle, alors qu'il faudrait le faire par rapport à la **circonférence** de la balle. Pour réaliser cela, il faut tenir compte du rayon de la balle (**ballRadius**). A vous de faire le nécessaire afin de donner les nouvelles implémentations des **collisions**.

Au passage, et pour montrer que vous avez bien suivi jusqu'ici, vous ferez en sorte de **changer la couleur de la balle** de façon **aléatoire** à chaque fois qu'elle **touche le bord** du **Canvas**. Après vous être assuré que votre code fonctionne, vous le mettrez en commentaires avant de passer à la suite.

Il nous reste à implémenter la **palette** ou **raquette** en définissant son **déplacement** avec les **touches du clavier**. Cette **palette** nous permettra de « *frapper* » la **balle**.
Pour cela, nous allons définir 3 nouvelles variables :

```
var paddleHeight = 10;
var paddleWidth = 100;
var paddleX = (canvas.width-paddleWidth) / 2;
```

Ici, nous définissons la **hauteur** et la **largeur** de la **palette**, ainsi que son point de départ sur l'axe **OX**, pour une utilisation dans les calculs ultérieurs dans le code. Créons maintenant une fonction qui dessinera la **palette** à l'écran.

```
function drawPaddle() {
 ctx.beginPath();
 ctx.rect(paddleX, canvas.height-paddleHeight, paddleWidth, paddleHeight);
 ctx.fillStyle = "#F50EB6";
 ctx.fill();
 ctx.closePath();
}
```

Nous allons maintenant implémenter le **déplacement** de cette **palette** à l'écran afin de permettre à l'utilisateur de la guider avec les **touches du clavier** et/ou **la souris**.

Nous aurons besoin de :

- Deux variables permettant de stocker des informations indiquant si le bouton de commande **gauche** ou **droit** est **enfoncé**.
- Deux **écouteurs d'événement** pour les évènements **keydown** et **keyup** - nous voulons exécuter du code pour gérer le mouvement de la palette lorsque les boutons sont **enfoncés**.
- Deux fonctions gérant les événements **keydown** et **keyup** et qui contiennent le code qui sera exécuté lorsque les boutons sont **enfoncés** ou **relâchés**.
- La possibilité de déplacer la **palette** à **gauche** et à **droite**

Les boutons **pressés** peuvent être **définis** et **initialisés** avec des **variables booléennes**, comme ceci.

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

Pour écouter les appuis sur les touches, nous allons configurer **deux écouteurs d'événement** pour les **touches claviers, un écouteur d'évènement** pour la **souris, pour** cela nous utilisons la méthode *incontournable* **JavaScript addEventListener())**. Ajoutez les lignes suivantes juste au-dessus de la méthode **setInterval(draw, 10)** :

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

Lorsque l'évènement **keydown** (touche enfoncée) est déclenché sur une des **touches de votre clavier**, la fonction **keyDownHandler()** est déclenchée.

Lorsque l'évènement **keyup** (touche relâchée) est déclenché sur une des touches de votre clavier, la fonction **keyUpHandler()** est déclenchée.
Maintenant, voici l'implémentation des fonctions précitées :

```
function keyDownHandler(e) {
    if(e.key == "Right" || e.key == "ArrowRight") {
        rightPressed = true;
    }else if(e.key == "Left" || e.key == "ArrowLeft") {
        leftPressed = true;
    }
}
function keyUpHandler(e) {
    if(e.key == "Right" || e.key == "ArrowRight") {
        rightPressed = false;
    }else if(e.key == "Left" || e.key == "ArrowLeft") {
        leftPressed = false;
    }
}
function mouseMoveHandler(e) {
    var relativeX = e.clientX - canvas.offsetLeft;
    if (relativeX > 0 && relativeX < canvas.width) {
        paddleX = relativeX - paddleWidth / 2;
     }
}
```

Lorsque nous appuyons sur une touche, ces informations sont stockées dans une variable. La variable pertinente dans chaque cas est définie sur **true**. Lorsque la *clé* est relâchée, la variable est remise à **false**.
Les deux fonctions prennent un événement en tant que paramètre, représenté par la variable **e**.
À partir de là, nous pouvons obtenir des informations utiles: la **key** contient les informations sur la touche sur laquelle nous avons appuyé. La plupart des navigateurs utilisent **ArrowRight** et **ArrowLeft** pour les touches de curseur **droite** / **gauche**, mais nous devons également inclure **Right** et **Left** contrôles pour soutenir les navigateurs **IE / Edge**. Si vous appuyez sur le curseur **gauche**, la variable **leftPressed** est définie sur **true**, et lorsqu'elle est relâchée, la variable **leftPressed** est définie sur **false**. Le même motif suit avec le curseur **droit** et la variable **rightPressed**.

Maintenant que nous avons les variables pour enregistrer les **actions** claviers nous pouvons mettre ces **variables** et **fonctions** en situation dans la fonction **draw()**.
Ainsi, la **raquette** se déplacera dans les limites imposées par le **Canvas**.

```
if(rightPressed && paddleX < canvas.width - paddleWidth) {
    paddleX += 8;
}else if(leftPressed && paddleX > 0) {
    paddleX -= 8;
}
```

Nous allons maintenant traiter le cas qui **achève la partie de jeu**. C'est-à-dire lorsque la **balle** touche le **bord inférieur** du **Canvas** (derrière la **raquette**).

Au lieu de laisser la **balle** rebondir sur les **quatre murs**, maintenant nous n'en autorisons plus que trois: **gauche, haut et droite**. Toucher *le mur* du **bas** mettra **fin à la partie**.
Retrouvez la condition qui permettait une **collision** avec le **bord inférieur** et nous allons modifier ce code pour qu'à chaque fois que la **balle** le touche la partie s'achève.
Pour le moment, nous allons rester simples en affichant **un message d'alerte** et en **redémarrant** le jeu en **rechargeant la page**.

Tout d'abord, retrouvez l'endroit où vous avez appelé initialement **setInterval()** et remplacez le code par celui-ci :

```
var interval = setInterval(draw, 10);
```

Puis modifions le code que nous avons repéré précédemment par celui-ci :

```
if(y + dy < ballRadius) {
   dy = -dy;
} else if(y + dy > canvas.height-ballRadius) {
   alert("GAME OVER");
   document.location.reload();
   clearInterval(interval);
}
```

Nous allons maintenant nous intéresser à la gestion de la **collision** entre la **balle** et la **raquette**.
La chose la plus simple à faire est de vérifier si **le centre de la balle** est entre **les bords gauches** et **droits** de la **raquette.**
Pour cela, nous devons encore modifier le code précédent par :

```
if(y + dy < ballRadius) {
   dy = -dy;
} else if(y + dy > canvas.height-ballRadius) {
    if(x > paddleX && x < paddleX + paddleWidth) {
      dy = -dy;
    }else {
      alert("GAME OVER");
      document.location.reload();
      clearInterval(interval);
    }
}
```

Ainsi, avant que la **balle** ne frappe le **bord inférieur** du **Canvas**, nous devons vérifier si elle ne frappe pas la **raquette**. Si oui, alors elle rebondit comme nous le souhaitions, sinon, le **jeu est terminé**.

Afin de rendre le jeu plus attrayant, il nous reste à définir des **briques,** à les **afficher** et à implémenter la **destruction des briques** qui seront **touchées** par la **balle**.

Nous allons définir le nombre de **rangées** et de **colonnes** de **briques**, leur **largeur** et leur **hauteur**, le **rembourrage** entre les briques afin qu'elles ne se touchent pas et un **décalage supérieur** et un **décalage gauche** afin qu'elles ne commencent pas à être tracées à partir du bord du **Canvas**.

```
var brickRowCount = 5;          // 5 Briques par ligne
var brickColumnCount = 3;       // 3 Briques par colonne
var brickWidth = 75;            // Largeur d'une Brique
var brickHeight = 20;           // Hauteur d'une Brique
var brickPadding = 10;          // Ecart entre les Briques
var brickOffsetTop = 30;        // Décalage supérieur
var brickOffsetLeft = 30;       // Décalage à gauche
```

Nous allons définir toutes nos **briques** dans un **tableau à deux dimensions**. Il contiendra les **colonnes de briques** (**c**), qui contiendront à leur tour les **rangées de briques** (**r**), qui contiendront chacune un **objet** contenant la position **x** et **y** permettant de **dessiner** chaque brique à l'écran. Ajoutez ce qui suit juste en dessous de vos variables:

```
var bricks = [];
for(var c=0; c<brickColumnCount; c++) {
    bricks[c] = [];
    for(var r=0; r<brickRowCount; r++) {
        bricks[c][r] = { x: 0, y: 0 };
    }
}
```

Les **objets** de **briques** seront utilisés ultérieurement pour la **détection** des **collisions** avec la **balle**.

Créons maintenant une fonction pour parcourir toutes les briques du tableau et les dessiner à l'écran. Notre code pourrait ressembler à ceci:

```
function drawBricks() {
    for(var c=0; c<brickColumnCount; c++) {
       for(var r=0; r<brickRowCount; r++) {
            bricks[c][r].x = 0;
            bricks[c][r].y = 0;
            ctx.beginPath();
            ctx.rect(0, 0, brickWidth, brickHeight);
            ctx.fillStyle = "#0095DD";
            ctx.fill();
            ctx.closePath();
       }
```

```
    }
}
```

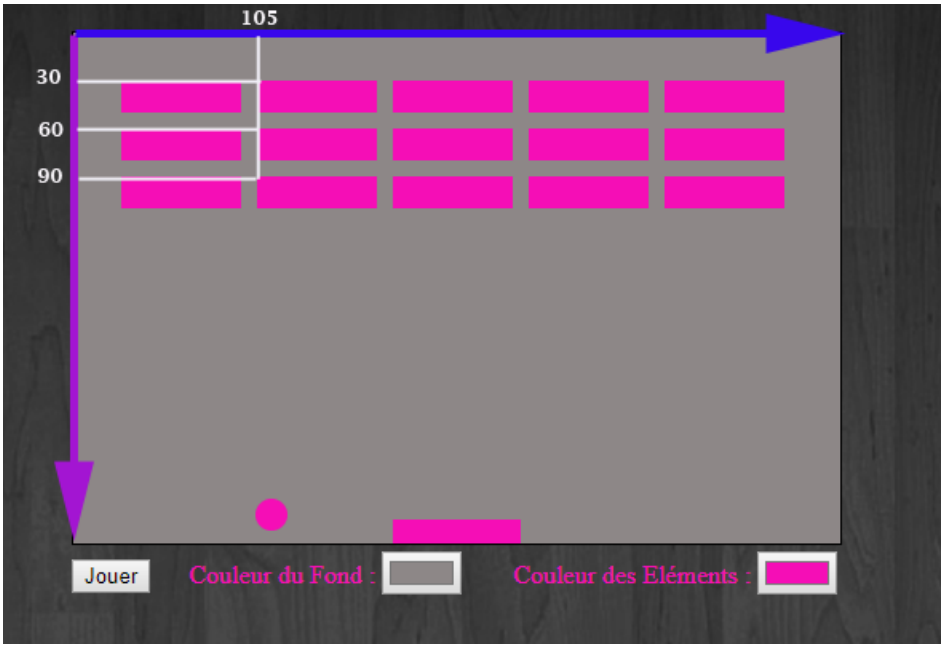
Seulement, nous dessinons ici des **briques** sur les **mêmes coordonnées** (0,0), il faut trouver une solution pour obtenir un placement des briques à chaque parcours des boucles, c'est-à-dire :

```
var brickX = (r * (brickWidth + brickPadding)) + brickOffsetLeft;
var brickY = (c * (brickHeight + brickPadding)) + brickOffsetTop;
```

Ce qui nous donnera pour par exemple la **deuxième colonne** (**r=1 et c dans {0,1,2}**) la représentation schématique qui va suivre. **brickX = 105 et brickY dans {30, 60, 90}**

De façon générale, pour chaque point de coordonnées **(brickX, brickY)** on tracera le rectangle suivant à partir de son point **supérieur gauche** de coordonnées **(brickX, brickY)** :

```
ctx.rect(brickX, brickY, brickWidth, brickHeight);
```



Ce qui nous permettra de modifier notre fonction **drawBricks()** en conséquence afin de dessiner nos briques dans le **Canvas,** avant de l'appeler dans la fonction **draw()** :

```
function drawBricks() {
    for (var c = 0; c < brickColumnCount; c++) {
        for (var r = 0; r < brickRowCount; r++) {
            var brickX = (r * (brickWidth + brickPadding)) + brickOffsetLeft;
            var brickY = (c * (brickHeight + brickPadding)) + brickOffsetTop;
            bricks[c][r].x = brickX;
            bricks[c][r].y = brickY;
```

```
            ctx.beginPath();
            ctx.rect(brickX, brickY, brickWidth, brickHeight);
            ctx.fillStyle = "#F50EB6";
            ctx.fill();
            ctx.closePath();
        }
    }
}
```

draw() …

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

…

Maintenant que nous avons dessiné notre **champ de briques** et que notre **balle est en mouvement**, nous allons nous intéresser à la détection des **collisions** entre la **balle** et les **briques**.

Pour cela, nous allons écrire une fonction de **détection de collisions** entre les **briques** et la **balle**.

Cette fonction va parcourir toutes les briques et va comparer la **position** de **chaque brique** aux **coordonnées** de la **balle** lorsque chaque image est dessinée. Pour une meilleure lisibilité du code, nous définirons la variable **b** permettant de **stocker l'objet brique** dans chaque **boucle de la détection de collision**:

```
function collisionDetection() {
    for (var c = 0; c < brickColumnCount; c++) {
        for (var r = 0; r < brickRowCount; r++) {
            var b = bricks[c][r];
            // Calculs
        }
    }
}
```

Si le **centre de la balle** se trouve à l'intérieur **des coordonnées d'une de nos briques**, nous changerons **la direction de la balle**. Pour que le centre de la balle soit à l'intérieur de la brique, les quatre affirmations suivantes doivent être vraies:

- La position x de la balle est supérieure à la position x de la brique.
- La position x de la balle est inférieure à la position x de la brique plus sa largeur.
- La position y de la balle est supérieure à la position y de la brique.
- La position y de la balle est inférieure à la position y de la brique plus sa hauteur.

Écrivons cela dans le code:

```
function collisionDetection() {
    for (var c = 0; c < brickColumnCount; c++) {
        for (var r = 0; r < brickRowCount; r++) {
            var b = bricks[c][r];
```

```
            if(x > b.x && x < b.x+brickWidth && y > b.y && y <
b.y+brickHeight) {
                dy = -dy;
            }
        }
    }
}
```

Avec ce code, la **balle** *rebondira* bien sur la **brique** après **collision** mais il va falloir implémenter la **disparition** de la **brique** après **collision** :

Nous pouvons le faire en **ajoutant** un **paramètre supplémentaire** pour indiquer si nous voulons ou non dessiner chaque brique à l'écran. Dans la partie du code où nous initialisons les briques, ajoutons une propriété **status** à chaque objet de brique. Mettez à jour la partie suivante du code comme indiqué par la ligne en surbrillance :

```
var bricks = [];
for (var c = 0; c < brickColumnCount; c++) {
  bricks[c] = [];
  for (var r = 0; r < brickRowCount; r++) {
      bricks[c][r] = { x: 0, y: 0 , status: 1};
  }
}
```

Nous allons ensuite vérifier la valeur de la propriété **status** de chaque brique dans la fonction **drawBricks()** avant de la *dessiner*. Si **status == 1**, **dessinez-la**, mais si **status == 0**, elle a été touchée par la balle et nous **l'effaçons**.

Il suffit de rajouter cette condition dans la fonction **drawBricks()**

```
function drawBricks() {
  for (var c = 0; c < brickColumnCount; c++) {
     for (var r = 0; r < brickRowCount; r++) {
        if (bricks[c][r].status == 1) {
           var brickX = (r * (brickWidth + brickPadding)) + brickOffsetLeft;
           var brickY = (c * (brickHeight + brickPadding)) + brickOffsetTop;
           bricks[c][r].x = brickX;
           bricks[c][r].y = brickY;
           ctx.beginPath();
           ctx.rect(brickX, brickY, brickWidth, brickHeight);
           ctx.fillStyle = "#F50EB6";
           ctx.fill();
           ctx.closePath();
        }
     }
  }
}
```

Maintenant, nous devons impliquer la propriété **status** de brique dans la fonction **collisionDetection()** : si la **brique est active** (son **statut est 1**), nous vérifierons si la collision se produit; Si une **collision se produit**, nous allons définir le **statut** de la brique donnée à **0** afin qu'elle ne soit pas dessinée à l'écran. Ce qui nous donnera :

```
function collisionDetection() {
   for (var c = 0; c < brickColumnCount; c++) {
      for (var r = 0; r < brickRowCount; r++) {
         var b = bricks[c][r];
         if(b.status == 1) {
            if(x > b.x && x < b.x+brickWidth && y > b.y && y <
b.y+brickHeight) {
               dy = -dy;
               b.status = 0;
            }
         }
      }
   }
}
```

Il suffit ensuite d'appeler cette fonction **collisionDetection()** dans la fonction **draw()**. Rechargez la page de votre navigateur et vous pourrez vérifier que nous avons bien le résultat attendu ;-)

Maintenant que notre **jeu est fonctionnel**, nous allons gérer le **score** et le **gain de la partie** ;-)

Vous avez besoin d'une variable pour enregistrer le **score**. Ajoutez ce qui suit dans votre **JavaScript**, après le reste de vos variables:

```
var score = 0;                    // Initialisation du score à 0
```

Il nous faut écrire maintenant la fonction d'écriture du **score** dans la **Canvas**, soit la fonction **drawScore()**.

```
function drawScore() {
   ctx.font = "16px Arial";
   ctx.fillStyle = "#F50EB6";
   ctx.fillText("Score: " + score, 8, 20);
}
```

Dessiner du **texte** dans le **Canvas** revient à dessiner une forme. La définition de la police est identique à celle des CSS: vous pouvez définir la taille et le type de police dans la méthode **font()**. Utilisez ensuite **fillStyle()** pour définir la couleur de la police et **fillText()** pour définir le texte qui sera placé sur le **Canvas** et son emplacement.

Le premier paramètre est le texte lui-même - le code ci-dessus indique le nombre actuel de points - et les deux derniers paramètres sont les **coordonnées** où le **texte** sera placé dans le **Canvas**.
Pour que le **score s'incrémente** à chaque fois qu'une **brique disparaît** il faut implémenter **score++** dans la fonction **collisionDetection().**
Pour terminer, il faut penser à appeler la fonction **drawScore()** dans la fonction **draw()**.

Maintenant, si le joueur arrive à faire disparaître toutes les briques il gagne la partie, il faut implémenter cela dans la fonction **detectionCollision()**.
…

```
if(score == brickRowCount*brickColumnCount) {
   alert("Félicitations, vous avez gagné !");
   document.location.reload();
   clearInterval(interval);
}
```

…
La fonction **document.location.reload()** recharge la page et redémarre le jeu lorsque l'utilisateur clique sur le bouton **Ok** de la **boîte d'alerte**.

Nous pouvons aussi donner des **vies** au joueur, à vous de gérer cette nouvelle fonctionnalité, donner **3 vies** au joueur **(lives == 3**). A chaque fois que la balle touche le **bord inférieur** du **Canvas**, il suffira de décrémenter la variable **lives** (**lives --**)

Afin d'améliorer le rendu, nous allons plutôt utiliser la méthode **requestAnimationFrame()** plutôt que **setInterval()**. La méthode **requestAnimationFrame()** aide le navigateur à rendre le jeu meilleur que **setInterval()** que nous utilisons actuellement. Cette méthode utilise une fréquence de **60 Hz** (60 images par seconde sont dessinées dans le **Canvas**).

Il suffit de remplacer la ligne :

```
var interval = setInterval(draw, 10);
```

par la ligne :

```
draw();
```

Puis, supprimer chaque instance de :

```
clearInterval(interval);
```

Puis, tout en bas de la fonction **draw()** (juste avant l'accolade fermante), ajoutez la ligne suivante, ce qui provoquera l'appel de la fonction **draw()** à plusieurs reprises:

```
requestAnimationFrame(draw);
```

La fonction **draw()** est maintenant exécutée maintes et maintes fois dans une boucle **requestAnimationFrame()**, mais au lieu de la fréquence d'images fixée à **10 millisecondes**, nous redonnons le contrôle du nombre d'images au **Navigateur**. Il synchronisera le framerate en conséquence et restituera les formes uniquement lorsque ce sera

nécessaire. Cela produit une boucle d'animation plus efficace et plus fluide que l'ancienne méthode **setInterval()**.
A ce niveau-là le jeu est complètement fonctionnel, vous allez apporter des améliorations en termes de présentation :

- Vous devrez d'une part, rendre fonctionnel le bouton lié au **changement de la couleur** du **fond** du **Canvas** puis le bouton qui permet de **changer la couleur** des **éléments** de l'interface utilisateur (libellés, titre, balle, briques, raquette).

- Vous implémenterez l'action sur le **bouton Jouer** afin qu'il permette à l'utilisateur de lancer une nouvelle partie.

- Vous devrez rendre **persistantes** ces informations ; en effet nous souhaitons que lorsque l'utilisateur quittera le jeu et lorsqu'il ouvrira une nouvelle session du jeu il devra retrouver ses derniers choix.
  Pour cela, vous pourrez utiliser dans un premier temps les **Cookies** avec **JavaScript**.
  Puis dans un deuxième temps vous explorerez la solution liée au **Local Storage** du Navigateur.

Maintenant que vous êtes des **pros** en développement de **jeux 2D** avec le **Canvas** de **HTML5** vous allez pouvoir développer vos propres jeux 2D.
Pour vous donner des idées, pourquoi pas vous fixer un challenge pour revisiter des **jeux 2D emblématiques** tels que : **Pong**, **Snake**, **Space Invaders**, **Galaga**, **Pac-Man**, **Bomb Jack**, **Frogger**, **Moon Cresta**, **Space Bomber**, **Wonder Boy**, …