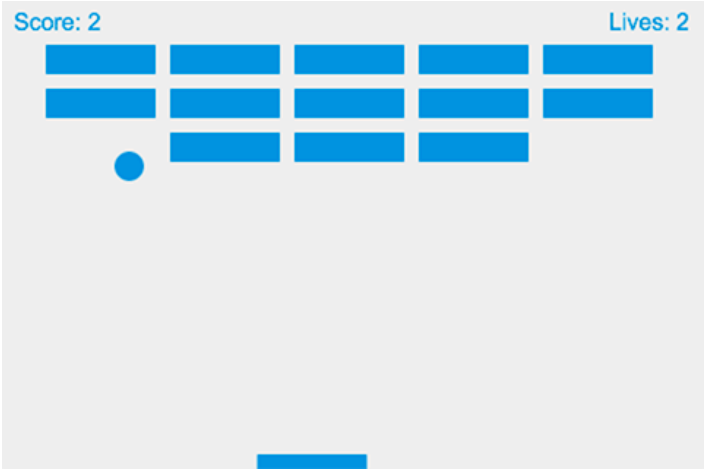
Jeu de casse-briques 2D en JavaScript

Dans ce tutoriel, nous allons créer pas à pas un jeu de casse-briques, créé entièrement avec JavaScript (sans framework), avec la balise HTML5 `<canvas>`.

Vous allez apprendre les bases d'utilisations de l'élément `<canvas>` pour implémenter des mécaniques de base du jeu vidéo, comme charger et déplacer des images, les détections de collisions, les mécanismes de contrôle, et les conditions de victoire/défaite.

Pour comprendre la plupart des articles de ce tutoriel, vous devez déjà avoir un niveau basique ou intermédiaire en JavaScript. À la fin de ce tutoriel, vous serez capable de créer vos propres jeux Web.



MENU

Commencer avec du Javascript pur et dur est le meilleur moyen d'acquérir des connaissances de développement de jeu web. Après ceci, vous pourrez prendre n'importe quel "framework" et l'utiliser pour vos projets. Les "frameworks" sont des outils créés avec le langage Javascript ; donc, même si vous voulez travailler avec ces derniers, c'est toujours bon d'apprendre le langage lui-même pour savoir ce qu'il se passe exactement. Les "frameworks" améliorent la vitesse de développement et aident à traiter les parties les moins intéressantes du jeu, mais si quelque chose ne fonctionne pas comme prévu, vous pouvez toujours essayer de déboguer ou juste écrire vos propre solutions en Javascript.

1 - Créer l'élément Canvas et l'afficher

Avant d'écrire les fonctionnalités de notre jeu, nous devons créer une structure où le jeu sera rendu. C'est possible en utilisant HTML et l'élément `<canvas>`.

La page HTML du jeu

La structure de la page HTML est vraiment simple, car tout le jeu sera contenu dans l'élément `<canvas>`. Avec votre éditeur de texte préféré, créez un nouveau fichier HTML, sauvegardez-le sous le nom `index.html`, et ajoutez-y le code suivant :

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="utf-8" />
		<title>Gamedev Canvas Workshop</title>
		<style>
			* { padding: 0; margin: 0; }
			canvas { background: #eee; display: block; margin: 0 auto; }
		</style>
	</head>
	<body>
		<canvas id="monJeux" width="480" height="320"></canvas>
		<script>
			// JavaScript code goes here
		</script>
	</body>
</html>
```

Dans l'en-tête, nous avons défini l'encodage des caractères (`charset`), le titre `<title>` et quelques règles CSS très simples. Le corps contient les éléments `<canvas>` et `<script>`. L'élément `<canvas>` contiendra le rendu du jeu et l'élément `<script>` l'emplacement du code JavaScript pour contrôler le jeu. L'élément `<canvas>` a un identifiant nommé `monJeux` qui permettra de le retrouver facilement en JavaScript, et possède des dimensions de 480 pixels de longueur et 320 pixels de hauteur. Tout le code JavaScript que nous allons écrire dans ce tutoriel sera contenu entre la balise ouvrante `<script>` et la balise fermante `</script>`.

Les bases de Canvas

Pour utiliser l'élément `<canvas>`, pour le rendu graphique de notre jeu, nous devons d'abord en donner la référence à l'interpréteur JavaScript. Ajoutez le code après la balise ouvrante `<script>`.

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

Ici nous avons enregistré la référence à l'élément `<canvas>` dans une variable nommée `canvas`. Ensuite, nous créons la variable ctx pour stocker le contexte de rendu 2D — l'outil réel que nous pouvons utiliser pour peindre sur Canvas.

Voyons un exemple de code qui imprime un carré rouge sur le canevas. Ajoutez ceci en dessous de vos lignes précédentes de JavaScript, puis chargez votre `index.html` dans un navigateur pour l'essayer.

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

Toutes les instructions sont entre les méthodes `beginPath()` et `closePath()` . Nous définissons un rectangle en utilisant `rect()` : les deux premières valeurs spécifient les coordonnées du coin supérieur gauche du rectangle tandis que les deux suivantes spécifient la largeur et la hauteur du rectangle. Dans notre cas, le rectangle est peint à 20 pixels du côté gauche de l'écran et à 40 pixels du haut, et a une largeur de 50 pixels et une hauteur de 50 pixels, ce qui en fait un carré parfait. La propriété `fillStyle` stocke une couleur qui sera utilisée par la méthode `fill()` pour peindre le carré en rouge.

Nous ne sommes pas limités aux rectangles, voici un code pour imprimer un cercle vert. Essayez d'ajouter ceci au bas de votre JavaScript, puis sauvegardez et rafraîchissez :

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

Comme nous pouvons le voir, nous utilisons à nouveau les méthodes `beginPath()` et `closePath()` . Entre elles, la partie la plus importante du code ci-dessus est la méthode `arc()` . Elle comporte six paramètres :

- les coordonnées `x` et`y` du centre de l'arc
- rayon de l'arc

- l'angle de départ et l'angle de fin (pour finir de dessiner le cercle, en radiant)
- direction du dessin (`false`*(faux)* pour le sens des aiguilles d'une montre, la valeur par défaut, ou `true` (vrai) pour le sens inverse). Ce dernier paramètre est facultatif.

La propriété `fillStyle` semble différente par rapport à l'exemple précédent. C'est parce que, tout comme avec CSS, la couleur peut être spécifiée sous la forme d'une valeur hexadécimale, d'un mot-clé, de la fonction `rgba ()` *(RVBA)* ou de toute autre méthode disponible pour les couleurs.

Au lieu d'utiliser `fillStyle` et de remplir les formes avec des couleurs, nous pouvons utiliser `stroke()` pour ne colorer que le contour exterieur. Essayez d'ajouter ce code à votre JavaScript aussi :

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

Le code ci-dessus affiche un rectangle vide avec des traits bleus. Grâce au canal alpha de la fonction rgba (), la couleur bleue est semi transparente.

Exercice 1 : changer la taille et la couleur des formes géométriques.

2 - Déplacer la balle

Nous avons vu dans précédemment comment dessiner une balle, maintenant déplaçons là. Techniquement, nous afficherons la balle sur l'écran, puis nous l'effacerons et ensuite nous la repeindrons dans une position légèrement différente et ceci à chaque image afin de donner l'impression d'un mouvement (tout comme le fonctionnement du mouvement dans les films).

Définir une boucle de dessin

Afin que le dessin soit mis à jour sur le canevas à chaque image, nous allons définir une fonction `draw()` qui sera exécutée en continu et qui utilisera des variables pour les positions des sprites, etc. Pour qu'une fonction s'exécute de façon répétée avec JavaScript, on pourra utiliser les méthodes `setInterval()` ou `requestAnimationFrame()`.

Supprimez tout le JavaScript que vous avez actuellement dans votre HTML à l'exception des deux premières lignes puis ajoutez ce qui suit en dessous de ces lignes. La fonction `draw()` sera exécutée toutes les 10 millisecondes (environ) 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, sans arrêt jusqu'à ce que nous y mettions un terme. Maintenant, dessinons la balle — ajoutons le code ci-dessous à notre fonction `draw()` :

```
ctx.beginPath();
ctx.arc(50, 50, 10, 0, Math.PI*2);
ctx.fillStyle = "#0095DD";
ctx.fill();
ctx.closePath();
```

Essayez votre code mis à jour maintenant, la balle devrait être repeinte sur chaque image.

Déplacer la balle

Pour le moment, vous ne voyez pas la balle "repeinte" car elle ne bouge pas. Améliorons tout ça. Pour commencer, au lieu d'une position bloquée à (50,50), nous allons définir un point de départ en bas et au milieu du canevas grâce aux variables `x` et `y` que nous utiliserons pour définir la position où le cercle est dessiné.

Ajoutez d'abord les deux lignes suivantes au-dessus de votre fonction `draw()` pour définir `x` et `y` :

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

Ensuite, mettez à jour la fonction `draw()` afin d'utiliser les variables x et y dans la méthode `arc()`, comme indiqué dans la ligne mise en évidence ci-dessous :

```
function draw() {
  ctx.beginPath();
  ctx.arc(x, y, 10, 0, Math.PI*2);
  ctx.fillStyle = "#0095DD";
  ctx.fill();
  ctx.closePath();
}
```

Nous voici à la partie importante : nous voulons ajouter une valeur à `x` et `y` après que chaque image ait été dessinée afin de faire croire que la balle bouge. On définit ces valeurs comme `dx` et `dy` avec comme valeurs respectives 2 et -2. Ajoutez le code après la déclaration des variables `x` et `y` :

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

La dernière chose à faire est de mettre à jour `x` et `y` avec nos variables `dx` et `dy` sur chaque image, de sorte que la balle sera peinte dans la nouvelle position à chaque nouvelle image. Ajoutez les deux nouvelles lignes, indiquées ci-dessous, à votre fonction `draw()` :

```
function draw() {
  ctx.beginPath();
  ctx.arc(x, y, 10, 0, Math.PI*2);
  ctx.fillStyle = "#0095DD";
  ctx.fill();
  ctx.closePath();
  x += dx;
  y += dy;
}
```

Enregistrez à nouveau votre code et essayez-le dans votre navigateur.

Vous devriez avoir le résultat suivant, ça fonctionne mais on a une trainée laissée par la balle derrière elle :

Effacer le canevas avant chaque image (*frame*)

La balle laisse une trace parce que qu'une nouveau cercle est dessiné sur chaque frame sans qu'on enlève le précédent. Pas d'inquiétude, il existe un moyen d'effacer le contenu du canevas : `clearRect()`. 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 sera effacée.

Ajoutez la nouvelle ligne en surbrillance ci-dessous à la fonction `draw()` :

```
function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.beginPath();
  ctx.arc(x, y, 10, 0, Math.PI*2);
  ctx.fillStyle = "#0095DD";
  ctx.fill();
  ctx.closePath();
  x += dx;
  y += dy;
}
```

Enregistrez votre code et essayez à nouveau. Cette fois, vous verrez la balle se déplacer sans laisser de trace. Toutes les 10 millisecondes, le canvas est effacé, la balle est dessinée sur une position donnée et les valeurs `x` et `y` sont mises à jour pour l'image suivante (en anglais, on parle de "*frame*").

Nettoyer votre code

Dans les prochains articles, nous allons ajouter de plus en plus de d'instructions à la fonction `draw()`. Mieux vaut donc la garder aussi simple et propre que possible. Commençons par déplacer le code s'occupant de dessiner la balle vers une fonction séparée.

Remplacez la fonction `draw()` existante par les deux fonctions suivantes :

```
function drawBall() {
  ctx.beginPath();
  ctx.arc(x, y, 10, 0, Math.PI*2);
  ctx.fillStyle = "#0095DD";
  ctx.fill();
  ctx.closePath();
}

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

Exercice 2 : changer la vitesse de la balle en mouvement et la direction dans laquelle elle se déplace.

3 - Faire rebondir la balle sur les murs

Notre balle bouge mais elle disparaît rapidement de l'écran. Pour y pallier, nous allons mettre en place une détection de collision pour faire rebondir la balle sur les quatre bords de l'espace de jeux.

Détection des collisions

Pour détecter la collision, nous vérifierons si la balle touche (entre en collision avec) les bords et, si c'est le cas, nous modifierons la direction de son mouvement en conséquence.

Pour faciliter les calculs, nous allons définir une variable appelée `ballRadius` qui contiendra le rayon du cercle dessiné et sera utilisée pour les calculs. Ajoutez cette variable à votre code, quelque part en dessous des déclarations de variables existantes :

```
var ballRadius = 10;
```

Mettez maintenant à jour la ligne qui dessine la balle à l'intérieur de la fonction `drawBall()` :

```
ctx.arc(x, y, ballRadius, 0, Math.PI*2);
```

Rebondir en haut et en bas

Il y a 4 bords en tout mais nous allons d'abord nous pencher sur le bord du haut. Nous devons, à chaque rafraichissement du canvas, regarder si la balle touche le bord du haut. Si c'est le cas, alors nous devons inverser la direction de la balle pour créer un effet de limite de zone de jeu. Il ne faut surtout pas oublier que le point d'origine est en haut à gauche ! Nous pouvons donc écrire :

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

Si la valeur y de la position de la balle est inférieure à zéro, changez la direction du mouvement sur l'axe y en le rendant égal à son inverse. Si la balle se déplaçait vers le haut à une vitesse de 2 pixels par image, elle se déplacera maintenant "vers le haut" à une vitesse de -2 pixels, ce qui équivaut en fait à se déplacer vers le bas à une vitesse de 2 pixels par image.

Le code ci-dessus traite du rebondissement de la balle sur le bord supérieur, alors traitons maintenant le bord inférieur :

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

Si la position en y de la balle est supérieure à la hauteur du canvas (soit 480 pixels dans cette leçon) on inverse encore la vitesse de la balle.

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

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

Si une des deux conditions est vérifiée, alors la vitesse est inversée. Essayez de créer votre propre code pour la gauche et la droite avant de passer à la prochaine sous-partie. Vous verrez que le principe est le même.

Rebondir à gauche et à droite

Nous avons couvert les bords supérieur et inférieur, alors pensons à ceux de gauche et de droite. C'est très similaire, il suffit de répéter les instructions pour `x` au lieu de `y` et sur la largeur (`width`) au lieu de la hauteur (`height`).

À ce stade, vous devez insérer les lignes de codes ci-dessus dans la fonction draw(), juste avant l'accolade de fermeture.

Testez votre code : nous avons maintenant une balle qui rebondit sur les quatre bords. Mais nous avons un autre problème : lorsque la balle frappe un mur, elle s'y enfonce légèrement avant de changer de direction :

C'est parce que nous calculons le point de collision entre le mur et le centre de la balle, alors que nous devrions le faire pour sa circonférence. La balle devrait rebondir juste après avoir touché le mur, et non pas lorsqu'elle est déjà à mi-chemin dans le mur, alors ajustons un peu nos déclarations pour inclure cela. Mettez à jour le dernier code que vous avez ajouté :

```
if(x + dx > canvas.width-ballRadius || x + dx < ballRadius) {
    dx = -dx;
}
if(y + dy > canvas.height-ballRadius || y + dy < ballRadius) {
    dy = -dy;
}
```

Lorsque la distance entre le centre de la balle et le bord du mur est exactement la même que le rayon de la balle, cela change la direction du mouvement. Soustraire le rayon de la largeur d'un bord et l'ajouter à l'autre nous donne l'impression d'une détection de collision correcte : la balle rebondit sur les bords comme elle devrait le faire.

Exercice 3 : changer la couleur de la balle à chaque fois que celle-ci rebondit sur un bord.

4 - Raquette et contrôle clavier

La balle rebondit librement partout indéfiniment mais il n'y a pas d'interaction avec le joueur. Ce n'est pas un jeu si vous ne pouvez pas le contrôler ! Nous allons donc ajouter une interaction avec le joueur : une ‘raquette’ contrôlable.

Créer une raquette pour frapper la balle

Il nous faut donc une raquette pour frapper la balle. Définissons quelques variables pour cela. Ajoutez les variables suivantes en haut de votre code, près de vos autres variables :

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

Ici, nous définissons la hauteur et la largeur de la raquette et son point de départ sur l'axe des x pour l'utiliser dans les calculs plus loin dans le code. Créons une fonction qui dessinera la raquette sur l'écran. Ajoutez ce qui suit juste en dessous de votre fonction `drawBall()` :

```
function drawPaddle() {
    ctx.beginPath();
    ctx.rect(paddleX, canvas.height-paddleHeight, paddleWidth, paddleHeight);
    ctx.fillStyle = "#0095DD";
    ctx.fill();
    ctx.closePath();
}
```

Permettre à l'utilisateur de contrôler la raquette

Nous pouvons dessiner la raquette où nous voulons, mais elle doit répondre aux actions de l'utilisateur. Il est temps de mettre en place certaines commandes au clavier. Nous aurons besoin de ce qui suit :

- Deux variables pour stocker des informations sur l'état des touches "gauche" et "droite".
- Deux ‘écouteurs’ d'événements pour les événements `keydown` et `keyup` du clavier. Nous voulons exécuter un code pour gérer le mouvement de la raquette lorsque des appuis sur les touches.
- Deux fonctions gérant les événements `keydown` et `keyup` et le code qui sera exécuté lorsque les touches sont pressées.
- La possibilité de déplacer la raquette vers la gauche et vers la droite

L'état des touches peut être mémorisé dans des variables booléennes comme dans l'exemple ci-dessous. Ajoutez ces lignes près de vos variables :

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

La valeur par défaut pour les deux est fausse car au début, car les touches ne sont pas enfoncés. Pour être informé des appuis sur les touches, nous allons mettre en place deux écouteurs d'événements. Ajoutez les lignes suivantes juste au-dessus de la ligne setInterval() au bas de votre JavaScript :

```
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 principe est vrai pour le deuxième écouteur : 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.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;
    }
}
```

Quand on presse une touche du clavier, l'information est stockée dans une variable. La variable concernée est mis sur `true`. Quand la touche est relachée, la variable revient à `false`.

Les deux fonctions prennent un événement comme paramètre, représenté par la variable `e`. De là, vous pouvez obtenir des informations utiles : la propriété `key` contient les informations sur la touche qui a été enfoncée. La plupart des navigateurs utilisent `ArrowRight` et `ArrowLeft` pour les touches de flèche gauche/droite, mais nous devons également tester `Right` and `Left` pour prendre en charge les navigateurs IE/Edge. Si la touche gauche est enfoncé, la variable `leftPressed` est mise à `true`, et lorsqu'elle est relâchée, la variable `leftPressed` est mise à `false`. Le même principe s'applique à la touche droite et à la variable `RightPressed`.

La logique du déplacement de la raquette

Nous avons maintenant mis en place les variables pour stocker les informations sur les touches pressées, les écouteurs d'événements et les fonctions associées. Ensuite, nous allons entrer dans le code pour utiliser tout ce que nous venons de configurer et pour déplacer la palette à l'écran. Dans la fonction `draw()`, nous vérifierons si les touches gauche ou droite sont pressées lors du rendu de chaque image. Notre code pourrait ressembler à ceci :

```
if(rightPressed) {
    paddleX += 7;
}
else if(leftPressed) {
    paddleX -= 7;
}
```

Si la touche gauche est enfoncée, la raquette se déplacera de sept pixels vers la gauche, et si la droite est enfoncé, la raquette se déplacera de sept pixels vers la droite. Cela fonctionne actuellement, mais la raquette disparaît du bord du canevas si nous maintenons l'une ou l'autre des touches trop longtemps enfoncée. Nous pourrions améliorer cela et déplacer la raquette uniquement dans les limites du canevas en changeant le code comme ceci :

```
if(rightPressed) {
    paddleX += 7;
    if (paddleX + paddleWidth > canvas.width){
        paddleX = canvas.width - paddleWidth;
    }
}
else if(leftPressed) {
    paddleX -= 7;
    if (paddleX < 0){
        paddleX = 0;
    }
}
```

La position de `paddleX` que nous utilisons variera entre `0` sur le côté gauche du canevas et `canvas.width-paddleWidth` sur le côté droit, ce qui fonctionnera exactement comme nous le souhaitons.

Ajoutez le bloc de code ci-dessus dans la fonction `draw()` en bas, juste au-dessus de l'accolade de fermeture.

Il ne reste plus qu'à appeler la fonction `drawPaddle()` depuis la fonction `draw()`, pour l'afficher réellement à l'écran. Ajoutez la ligne suivante à l'intérieur de votre fonction `draw()`, juste en dessous de la ligne qui appelle `drawBall()` :

```
drawPaddle();
```

<u>Exercice 4 :</u> faire aller la raquette plus vite ou plus lentement, changer sa taille et sa couleur.

5 - Fin de partie

La façon de perdre dans le casse briques est simple. Si vous ‘rattez’ la balle avec le paddle (la raquette) et qu’elle atteint le bas de l'écran, la partie est terminée.

Intégrer une fin de partie

Essayons d'intégrer une fin de partie dans le jeu . Voyons une partie du code de la troisième leçon, où nous faisions rebondir la balle contre les murs :

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

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

Au lieu de permettre à la balle de rebondir sur les quatre murs, nous n'en autoriserons que trois désormais gauche, haut et droite. Toucher le mur du bas mettra fin à la partie.

Nous allons donc modifier le second bloc `if` (qui gère le déplacement sur l'axe vertical, y) en y ajoutant un `else if` qui déclenchera un Game Over si la balle entre en collision avec le mur du bas. Pour l'instant nous allons rester simple, afficher un message d'alerte et redémarrer le jeu en rechargeant la page.

Tout d'abord remplacer l'appel initial à `setInterval()`

```
setInterval(draw, 10);
```

par

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

Puis remplacez la seconde instruction `if` par le code suivant:

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

Faire rebondir la balle sur la raquette

La dernière chose à faire dans cette leçon est de créer une sorte de détection de collision entre la raquette et la balle, de sorte qu'elle puisse rebondir et revenir dans la zone de jeu. La chose la plus facile à faire est de vérifier si le centre de la balle se trouve entre les bords droit et gauche du paddle. Mettez à jour le dernier bout de code que vous venez de modifier, comme-ci dessous :

```
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);
    }
}
```

Si la balle entre en collision avec le mur du bas, nous devons vérifier si elle touche la raquette. Si c'est le cas, la balle rebondit et revient dans la zone de jeu. Sinon, le jeu est terminé comme avant.

<u>Exercice 5 :</u> Faire en sorte que la balle accélère quand elle touche la raquette.

6 - Créer les briques

Ce dont a besoin un jeu de casse-brique ce sont des briques à détruire avec la balle. C'est ce que nous allons faire maintenant.

Mettre en place les variables "Brique"

Le principal objectif de cette leçon est d'avoir quelques lignes de code pour afficher les briques, en utilisant une boucle imbriquée qui va parcourir un tableau à deux dimensions. Cependant nous avons besoin de définir quelques variables pour stocker des informations décrivant les briques, telles que leur largeur, leur hauteur, les colonnes et les lignes, etc. Ajoutez les lignes suivantes dans votre code, sous les variables préalablement déclarées.

```
var brickRowCount = 3;
var brickColumnCount = 5;
var brickWidth = 75;
var brickHeight = 20;
var brickPadding = 10;
var brickOffsetTop = 30;
var brickOffsetLeft = 30;
```

Ici nous avons défini dans l'ordre le nombre de lignes et de colonnes de briques, mais également une hauteur, une largeur et un espacement (*padding*) entre les briques pour qu'elles ne se touchent pas entre elles et qu'elles ne commencent pas a être tracées sur le bord du canevas.

Nous allons placer nos briques dans un tableau à deux dimensions. Il contiendra les colonnes de briques (c), qui à leur tour contiendront les lignes de briques (r) qui chacune contiendront un objet défini par une position `x` et `y` pour afficher chaque brique sur l'écran.
Ajoutez le code suivant juste en-dessous des 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 };
    }
}
```

Le code ci-dessus va parcourir les lignes et les colonnes et créer de nouvelles briques. REMARQUE : les objets briques seront également utilisés plus tard afin de détecter les collisions.

Logique de dessin des briques

Maintenant créons une fonction pour parcourir toutes les briques dans le tableau et les dessiner sur l'écran. Notre code pourrait ressembler à ça :

```
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();
        }
    }
}
```

Une nouvelle fois, nous parcourons les colonnes et les lignes pour attribuer une position `x` et `y` à chaque brique, et nous dessinons les briques — de taille : `brickWidth` x `brickHeight` — sur le canevas, pour chaque itération de la boucle. Le problème est que nous les affichons toutes au même endroit, aux coordonnées `(0,0)`. Ce dont nous avons besoin d'inclure ce sont quelques calculs qui vont définir la position `x` et `y` de chaque brique à chaque passage dans la boucle :

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

Chaque position `brickX` est déterminée par `brickWidth` + `brickPadding`, multiplié par le nombre de colonnes, `c`, plus `brickOffsetLeft`; la logique pour `brickY` est identique excepté qu'on utilise pour les ligne les valeurs `r`,`brickHeight`

et `brickOffsetTop`. Maintenant chaque brique peut être dessinée à la bonne place - en lignes et colonnes, avec un espacement entre les briques, avec un espace par rapport à la gauche et au haut du contour du canvas.

La version finale de la fonction `drawBricks()`, après avoir assigné les valeurs `brickX` et `brickY` comme coordonnées, plutot que `(0,0)` à chaque fois, va ressembler à ceci — ajouter la fonction ci-dessous après `drawPaddle()` :

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

Afficher les briques

La dernière chose à faire dans cette leçon est d'ajouter un appel à `drawBricks()` quelque part dans la fonction `draw()`, préférablement au début, entre le nettoyage du canevas et le dessin de la balle. Ajoutez la ligne suivante juste en dessous de `drawBall()` :

```
drawBricks();
```

Exercice 6 : essayez de changer le nombre de briques dans une colonne ou dans une ligne ou bien leur position.

7 - Détection de collisions

Nous devons ajouter une détection des collisions afin qu'elle puisse rebondir sur les briques et les casser. Dans l'intérêt de ce tutoriel, nous le ferons de la manière la plus simple possible. Nous vérifierons si le centre de la balle entre en collision avec l'une des briques données. Cela ne donnera pas toujours un résultat parfait, et il existe des moyens beaucoup plus sophistiqués de détecter des collisions, mais cela fonctionnera assez bien pour vous apprendre les concepts de base.

Une fonction de détection de collision

Pour commencer, nous voulons créer une fonction de détection de collision qui va parcourir toutes les briques et comparer la position de chaque brique avec les coordonnées de la balle lorsque chaque image est dessinée. Pour une meilleure lisibilité du code, nous allons définir la variable `b` pour stocker l'objet brique dans la 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.

Écrivez cela sous forme de code:

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

Ajoutez le bloc ci-dessus à votre code, sous la fonction `keyUpHandler()` .

Faire disparaître les briques après qu'elles aient été touchées

Le code ci-dessus fonctionnera comme vous le souhaitez et la balle changera de direction. Le problème est que les briques restent là où elles sont. Nous devons trouver un moyen de nous débarrasser de celles que nous avons déjà touchées avec la balle. Nous pouvons le faire en ajoutant un paramètre supplémentaire pour indiquer si nous voulons ou non afficher chaque brique à l'écran. Dans la partie du code où nous initialisons les briques, ajoutons une propriété `status` à chaque brique. Mettez à jour la partie suivante du code comme indiqué par la ligne en évidence:

```
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 vérifierons ensuite la valeur de la propriété `status` de chaque brique dans la fonction `drawBricks()` avant de la dessiner. Si `status` vaut `1`, dessinez-la, mais s'il vaut `0`, la balle a été touchée et nous ne voulons pas la voir sur l'écran. Mettez à jour votre fonction `drawBricks()` comme suit:

```
function drawBricks() {
    for(var c=0; c<brickColumnCount; c++) {
        for(var r=0; r<brickRowCount; r++) {
            if(bricks[c][r].status == 1) {
                var brickX = (c*(brickWidth+brickPadding))+brickOffsetLeft;
                var brickY = (r*(brickHeight+brickPadding))+brickOffsetTop;
                bricks[c][r].x = brickX;
                bricks[c][r].y = brickY;
                ctx.beginPath();
                ctx.rect(brickX, brickY, brickWidth, brickHeight);
                ctx.fillStyle = "#0095DD";
                ctx.fill();
                ctx.closePath();
            }
        }
    }
}
```

Suivi et mise à jour de l'état dans la fonction de détection de collision

Nous devons maintenant impliquer la propriété de `status` de brique dans la fonction `collisionDetection()`: si la brique est active (son statut est `1`), nous vérifierons si une collision a lieu ; Si une collision se produit, nous allons définir l'état de la brique donnée sur `0` afin qu'elle ne soit pas affichée à l'écran. Mettez à jour votre fonction `collisionDetection()` comme indiqué ci-dessous:

```
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( ……) {
                    dy = -dy;
                    b.status = 0;
```

```
                }
            }
        }
    }
}
```

Activer notre détection de collision

La dernière chose à faire est d'ajouter un appel à la fonction `collisionDetection ()` à notre fonction `draw()` principale. Ajoutez la ligne suivante à la fonction `draw()`, juste en dessous de l'appel `drawPaddle()`:

```
collisionDetection();
```

Exercice 7: changez la couleur de la balle lorsqu'elle frappe une brique.

8 - Suivre le score et gagner

Calculer le score

Si vous pouvez voir votre score durant le jeu, vous pourrez impressioner vos amis. Vous avez besoin d'une variable pour stocker le score. Ajoutez ce qui suit dans votre JavaScript après le reste de vos variables :

```
var score = 0;
```

Vous avez aussi besoin d'une fonction `drawScore()`, pour créer et mettre à jour l'affichage du score. Ajoutez ce qui suit après la fonction de détection de collision `collisionDetection()`:

```
function drawScore() {
    ctx.font = "16px Arial";
    ctx.fillStyle = "#0095DD";
    ctx.fillText("Score: "+score, 8, 20);
}
```

Dessiner du texte sur un canvas revient à dessiner une forme. La définition de la police est identique à celle en CSS — vous pouvez définir la taille et le type avec la méthode `font()`. Puis utilisez `fillStyle()` pour définir la couleur de la police et `fillText()` pour définir la position du texte sur le canevas. 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 est placé sur le canevas.

Pour attribuer le score à chaque collision avec une brique, ajoutez une ligne à la fonction `collisionDetection()` afin d'incrémenter la valeur de la variable score à chaque détection d'une collision. Ajoutez à votre code la ligne mise en évidence ci-dessous :

```
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( ……) {
                    dy = -dy;
                    b.status = 0;
                    score++;
                }
            }
        }
    }
}
```

Appelez la fonction `drawScore()` dans la fonction `draw()` pour garder le score à jour à chaque nouvelle frame — ajoutez la ligne suivante dans la fonction `draw()`, en dessous de l'appel à `drawPaddle()` :

```
drawScore();
```

Ajoutez un message de victoire lorsque toutes les briques ont été détruites

Le comptage des points fonctionne bien, mais vous ne les compterez pas indéfiniment. Vous devez donc afficher un message de victoire si toutes les briques ont été détruites. Ajoutez la section mise en évidence dans votre fonction `collisionDetection()`:

```
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( ……) {
                    dy = -dy;
                    b.status = 0;
                    score++;
                    if(score == brickRowCount*brickColumnCount) {
                        alert("C'est gagné, Bravo!");
                        document.location.reload();
                        clearInterval(interval); // Needed for Chrome to end game
                    }
                }
            }
        }
    }
}
```

Grâce à ça, les utilisateurs peuvent réellement gagner le jeu. La fonction `document.location.reload()` recharge la page et redémarre le jeu au clic sur le bouton d'alerte.

Exercice 8 : Ajouter plus de points par brique touchée et indiquez le nombre de points gagnés dans la boîte d'alerte de fin de partie.

9 - Suivre le score et gagner

Calculer le score

Si vous pouvez voir votre score durant le jeu, vous pourrez impressioner vos amis. Vous avez besoin d'une variable pour stocker le score. Ajoutez ce qui suit dans votre JavaScript après le reste de vos variables :

```
var score = 0;
```

Vous avez aussi besoin d'une fonction `drawScore()`, pour créer et mettre à jour l'affichage du score. Ajoutez ce qui suit après la fonction de détection de collision `collisionDetection()`:

```
function drawScore() {
    ctx.font = "16px Arial";
    ctx.fillStyle = "#0095DD";
    ctx.fillText("Score: "+score, 8, 20);
}
```

Dessiner du texte sur un canvas revient à dessiner une forme. La définition de la police est identique à celle en CSS — vous pouvez définir la taille et le type avec la méthode `font()`. Puis utilisez `fillStyle()` pour définir la couleur de la police et `fillText()` pour définir la position du texte sur le canevas. 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 est placé sur le canevas.

Pour attribuer le score à chaque collision avec une brique, ajoutez une ligne à la fonction `collisionDetection()` afin d'incrémenter la valeur de la variable score à chaque détection d'une collision. Ajoutez à votre code la ligne mise en évidence ci-dessous :

```
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( ……) {
                    dy = -dy;
                    b.status = 0;
                    score++;
                }
            }
        }
    }
}
```

Appelez la fonction `drawScore()` dans la fonction `draw()` pour garder le score à jour à chaque nouvelle frame — ajoutez la ligne suivante dans la fonction `draw()`, en dessous de l'appel à `drawPaddle()` :

```
drawScore();
```

Ajoutez un message de victoire lorsque toutes les briques ont été détruites

Le comptage des points fonctionne bien, mais vous ne les compterez pas indéfiniment. Alors qu'en est-il du score lorsque toutes les briques ont été détruites ? Après tout c'est l'objectif principal du jeu. Vous devez donc afficher un message de victoire si toutes les briques ont été détruites. Ajoutez la section mise en évidence dans votre fonction `collisionDetection()`:

```
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( ……) {
                    dy = -dy;
                    b.status = 0;
                    score++;
                    if(score == brickRowCount*brickColumnCount) {
                        alert("C'est gagné, Bravo!");
                        document.location.reload();
                        clearInterval(interval); // Needed for Chrome to end game
                    }
                }
            }
        }
    }
}
```

Grâce à ça, les utilisateurs peuvent réellement gagner le jeu. La fonction `document.location.reload()` recharge la page et redémarre le jeu au clic sur le bouton d'alerte.

Exercice 9 : Ajoutez plus de points par brique touchée et indiquez le nombre de points gagnés dans la boîte d'alerte de fin de partie.

10 - Contrôle à la souris

Le jeu lui-même est terminé. Nous avons déjà ajouté des commandes au clavier, mais nous pourrions facilement ajouter des commandes à la souris.

Détecter les mouvements de la souris

Il est encore plus facile de détecter les mouvements de la souris que les pressions sur les touches : il suffit d'écouter l'évènement `mousemove`. Ajouter la ligne suivante au même endroit que les autres écouteurs d'événement, juste en dessous de `l'évènement keyup` :

```
document.addEventListener("mousemove", mouseMoveHandler, false);
```

Lier le mouvement de la raquette au mouvement de la souris

Nous pouvons mettre à jour la position de la raquette en fonction des coordonnées du pointeur — c'est exactement ce que fera la fonction de manipulation suivante. Ajoutez la fonction ci-dessous à votre code, sous la dernière ligne que vous avez ajoutée :

```
function mouseMoveHandler(e) {
    var relativeX = e.clientX - canvas.offsetLeft;
    if(relativeX > 0 && relativeX < canvas.width) {
        paddleX = relativeX - paddleWidth/2;
    }
}
```

Dans cette fonction, nous calculons d'abord une valeur `relativeX`, qui est égale à la position horizontale de la souris dans la fenêtre de visualisation (`e.clientX`) moins la distance entre le bord gauche de la toile et le bord gauche de la fenêtre de visualisation (`canvas.offsetLeft`) — en fait, cette valeur est égale à la distance entre le bord gauche du canevas et le pointeur de la souris. Si la position relative du pointeur X est supérieure à zéro et inférieure à la largeur du canevas, le pointeur se trouve dans les limites du canevas, et la position `paddleX` (ancrée sur le bord gauche de la palette) est fixée à la valeur `relativeX` moins la moitié de la largeur de la palette, de sorte que le mouvement sera en fait relatif au milieu de la raquette.

La raquette suivra désormais la position du curseur de la souris, mais comme nous limitons le mouvement à la taille du canevas, elle ne disparaîtra pas complètement d'un côté ou de l'autre.

Exercice 10 : ajustez les limites du mouvement de la raquette, de sorte que la raquette entière soit visible sur les deux bords du canevas au lieu de seulement la moitié.

11 - Finitions

Donner des vies au joueur

Mettre en œuvre des vies est assez simple. Ajoutons d'abord une variable pour stocker le nombre de vies à l'endroit où nous avons déclaré nos autres variables :

```
var lives = 3;
```

L'affichage du compteur de vie est similaire à celui du compteur de points — ajoutez la fonction suivante à votre code, sous la fonction `drawScore()` :

```
function drawLives() {
    ctx.font = "16px Arial";
    ctx.fillStyle = "#0095DD";
    ctx.fillText("Lives: "+lives, canvas.width-65, 20);
}
```

Au lieu de mettre immédiatement fin au jeu, nous allons réduire le nombre de vies jusqu'à ce qu'il n'y en ait plus. Nous pouvons également réinitialiser les positions du ballon et de la raquette lorsque le joueur commence sa prochaine vie. Ainsi, dans la fonction `draw()`, remplacez les trois lignes suivantes :

```
alert("GAME OVER");
document.location.reload();
clearInterval(interval); // Needed for Chrome to end game
```

Nous pouvons ainsi y ajouter une logique un peu plus complexe, comme indiqué ci-dessous :

```
lives--;
if(!lives) {
    alert("GAME OVER");
    document.location.reload();
    clearInterval(interval); // Needed for Chrome to end game
}
else {
    x = canvas.width/2;
    y = canvas.height-30;
    dx = 2;
    dy = -2;
    paddleX = (canvas.width-paddleWidth)/2;
}
```

Maintenant, quand la balle frappe le bord inférieur de l'écran, nous soustrayons une vie de la variable `lives`. S'il n'y a plus de vies, la partie est perdue ; s'il reste encore des vies, alors la position de la balle et la raquette sont remises à zéro, ainsi que le mouvement de la balle.

Afficher le compteur de vies

Maintenant, vous devez ajouter un appel à `drawLives()` dans la fonction `draw()` et l'ajouter sous l'appel `drawScore()`.

```
drawLives();
```

Améliorer le rendu avec requestAnimationFrame()

Maintenant, travaillons sur quelque chose qui n'est pas lié à la mécanique du jeu, mais à la façon dont il est rendu. `requestAnimationFrame` aide le navigateur à rendre le jeu mieux que la cadence fixe que nous avons actuellement mise en place en utilisant `setInterval()`. Remplacez la ligne suivante :

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

avec simplement :

```
draw();
```

et supprimez chaque occurence de :

```
clearInterval(interval); // Needed for Chrome to end game
```

Ensuite, tout en bas de la fonction `draw()` (juste avant l'accolade de fermeture), ajoutez la ligne suivante, ce qui fait que la fonction `draw()` s'appelle encore et encore :

```
requestAnimationFrame(draw);
```

La fonction `draw()` est maintenant exécutée indéfiniment dans une boucle `requestAnimationFrame()`, mais au lieu de la cadence fixe de 10 millisecondes, nous redonnons le contrôle de la cadence au navigateur. Il synchronisera la cadence en conséquence et ne n'acutalisera l'affichage que lorsque cela sera nécessaire. Cela permet d'obtenir une boucle d'animation plus efficace et plus fluide que l'ancienne méthode `setInterval()`.

Exercise 11: changer le nombre de vies et l'angle de rebond de la balle sur la raquette.