

Tutoriel pour l'introduction à l'animation en HTML5 et JavaScript à l'aide de la librairie JavaScript Box2D

Frédéric Guégan – Olivier Fauvel-Jaeger – Giacomo Rombaut

Table des matières

1.	Préparation du projet.....	2
2.	Début du projet : Création du monde et des objets	2
3.	Dessiner et mettre à jour votre animation	4
4.	Objets avancés	7
5.	Articuler les objets entre eux.....	9
6.	Appliquer des forces aux objets.....	11
7.	Gérer les collisions	12
8.	Aller plus loin	14

Ce tutoriel a pour but d'aider les étudiants et les professeurs afin de pouvoir produire une animation en HTML5 et JavaScript à l'aide de la librairie JavaScript Box2D.

Ce tutoriel va principalement aborder l'usage du JavaScript dans l'environnement d'un navigateur Web, il est donc de rigueur que vous sachiez coder à la fois en HTML et en CSS.

1. Préparation du projet

Vous trouverez ci-joint avec ce tutoriel un dossier contenant les fichiers nécessaires afin de pouvoir développer votre application. Ce dossier contient les deux fichiers JavaScript nécessaires à la librairie Box2D ainsi qu'un fichier html de base contenant un canvas vide et l'appel à la librairie.

Pour commencer votre projet vous devez recopier ce répertoire dans votre répertoire de travail. Puis créez un fichier JavaScript vide qui va contenir le début du code nécessaire à l'application.

2. Début du projet : Création du monde et des objets

Tout ce que nous allons définir ici sera dans une fonction init qui sera notre programme principal mais libre à vous de définir des fonctions externes puis de les appeler par la suite dans votre fonction principale. Cette fonction est appelée dans notre fichier html lors du chargement de la page.

Pour commencer nous avons besoin de définir un 'world' avec deux paramètres : la gravité et "sleep".

```
world = new b2World(  
    new b2Vec2(0, 10),  
    true  
);
```

Ici, nous définissons la gravité comme un vecteur égal à 10 unités dans la direction y, où (0,0) est en haut à gauche et y vers le bas. De plus, le paramètre "sleep" dit à Box2D de ne pas traiter l'objet si il est en équilibre.

Nous devons par la suite ajouter un autre paramètre important L'échelle qui fait correspondre les distances dans le monde Box2d (en mètres) à la distance correspondante sur notre toile (en pixels).

```
var scale = 20.0;
```

Ici, nous avons donc défini 20m = 1px.

Maintenant nous devons définir un sol pour notre monde :

Pour cela, nous définissons d'abord les propriétés (densité, friction, restitution) du sol. Ensuite nous définissons son corps comme étant "statique" (Box2D dispose de 3 types de corps : statique, dynamique et cinématique, reportez à ce manuel : <http://www.box2d.org/manual.html#Toc258082973> pour plus de détail). Ce sol sera représenté par un corps rectangulaire (un corps rectangulaire est défini par son centre, sa largeur et son hauteur). Puis, nous positionnons notre corps à la hauteur screenH(auteur) et à la moitié de screenL(argeur) et nous divisons ces valeurs par 'scale' afin de correspondre aux coordonnées Box2D. Pour finir, nous définissons notre sol comme étant un polygone que l'on dessine comme une boîte de largeur screenL et de hauteur 10, puis nous l'ajoutons à notre monde.

Ceci est résumé à travers le code suivant :

```
// Define the Ground
// Basic properties of ground
var fixDef = new b2FixtureDef;
fixDef.density = 2.0;
fixDef.friction = 0.9;
fixDef.restitution = 0.8;

// Ground is simply a static rectangular body with its center at screenW/2
// and screenH
var bodyDef = new b2BodyDef;
bodyDef.type = b2Body.b2_staticBody;
bodyDef.position.x = screenL/2/scale;
// We use screenH for y coordinate as the ground has to be at the bottom of
// our screen
bodyDef.position.y = screenH/scale;

// here we define ground as a rectangular box of width = screenW
// and height = 10 (just some small number to make a thin strip placed at
// the bottom)
fixDef.shape = new b2PolygonShape;
fixDef.shape.SetAsBox(screenL/scale, 10/scale);

// And finally add our ground object to our world
world.CreateBody(bodyDef).CreateFixture(fixDef);
```

Voyons maintenant comment définir des objets :

```
// Adding objects to our simulation space.
// The difference here being, that these are dynamic objects and are
// affected by forces and impulses
bodyDef.type = b2Body.b2_dynamicBody;
for(var i = 0; i < 10; ++i)
{
    if(Math.random() < 0.5) {
        fixDef.shape = new b2PolygonShape;
        fixDef.shape.SetAsBox(Math.random() + 0.5, Math.random() + 0.5);
    } else {
        fixDef.shape = new b2CircleShape(Math.random() + 0.5);
    }
    bodyDef.position.x = Math.random() * screenW/scale;
    bodyDef.position.y = Math.random() * screenH/scale/4;
    world.CreateBody(bodyDef).CreateFixture(fixDef);
}
```

La première chose importante à noter ici est le changement du type de corps pour 'dynamique' car nous allons avoir des objets qui vont interagir les uns avec les autres en vertu des lois de la physique. Le code ci-dessus ajoute au hasard rectangles et des cercles dans le monde en les plaçant au hasard sur toute la largeur et dans le quart supérieur de la toile.

3. Dessiner et mettre à jour votre animation

Maintenant que nous avons créé notre monde et nos premiers objets, il faut pouvoir les dessiner et les animer.

En premier, nous avons besoin d'une fonction de rappel. Il est largement conseillé d'utiliser `requestAnimationFrame` au lieu de `setTimeout` ou `setInterval` pour une meilleure performance.

```
window.requestAnimationFrame = (function() {
    return window.requestAnimationFrame ||
        window.webkitRequestAnimationFrame ||
        window.mozRequestAnimationFrame ||
        window.oRequestAnimationFrame ||
        window.msRequestAnimationFrame ||
        function(/* function */ callback, /* DOMElement */ element) {
            window.setTimeout(callback, 1000 / 60);
        };
})();
```

Voici maintenant notre fonction de mise à jour :

```
// Our update function
function update() {
    world.Step(1 / 60, 3, 3);

    if(drawDebug)
    {
        // Clears the canvas context
        context.clearRect(0,0,screenW, screenH);

        world.DrawDebugData();
        drawCanvasObjects();
    }
    else
        world.DrawDebugData();

    // This is called after we are done with time steps to clear the forces
    world.ClearForces();

    // callback for next update
    requestAnimationFrame(update);
};
```

Cette fonction dit à notre monde de se mettre à jour en calculant les forces, leur impact et les positions résultantes des objets dans le monde. Après cela, elle dessine le monde à l'aide de la fonction DrawDebugData (définie en dessous), une fonction native de Box2D utile lors du débogage de l'application. « if (drawDebug) » vérifie si vous souhaitez remplacer le rendu de base box2d avec des éléments HTML5. Dans cet exemple, je remplis les cercles en rouge. Il ne nous reste qu'à appeler notre fonction requestAnimationFrame(update) afin de démarrer l'animation.

```
var debugDraw = new b2DebugDraw();
debugDraw.SetSprite(context);
debugDraw.SetDrawScale(scale);
debugDraw.SetFillAlpha(0.5);
debugDraw.SetLineThickness(1.0);
debugDraw.SetFlags(b2DebugDraw.e_shapeBit | b2DebugDraw.e_jointBit |
b2DebugDraw.e_centerOfMassBit);

world.SetDebugDraw(debugDraw);

// debugDraw => defining DrawDebugData()<
// Function to detect circles and color them Red
function drawCanvasObjects()
{
    // get list of all bodies attached to this world
    var node = world.GetBodyList();
    while(node)
    {
        var curr = node;
        node = node.GetNext();
    }
}
```

```

// Check if the identified body is of type dynamic. Remember we defined
our circles as dynamic body types
if(curr.GetType() == b2Body.b2_dynamicBody)
{
    // Get the shape from the list of retrieved fixtures defined during
initialization for each body
    var shape = curr.GetFixtureList().GetShape();

    if(shape.GetType() == circle.GetType()) // If shape is circle
    {
        // Get the body's position in the world
        var position = curr.GetPosition();
        // Scale back the position to map them canvas coordinates
        var canvasY = position.y*scale;
        var canvasX = position.x*scale;

        // boundary color = white
        context.strokeStyle = "#000000";
        // fill color = red
        context.fillStyle = "#FF0000";

        context.beginPath();
        context.arc(canvasX,canvasY,shape.GetRadius()*scale,0,Math.PI*2,true);
        context.closePath();
        context.stroke();
        context.fill();
    }
}
}
}

```

Nous venons de finir les bases de Box2D, vous pouvez retrouver cet exemple dans le dossier fourni sous le nom "demoTutoriel.html" où j'ai ajouté deux bords sur le côté afin que les objets ne sortent pas de la zone ainsi qu'un debug plus avancé que celui présenté ici.

4. Objets avancés

Maintenant que nous avons vu comment créer des objets simples voyons comment définir des polygones plus avancés à l'aide de Box2D.

Créons par exemple un triangle : Pour cela, il faut définir les différents côtés de notre triangle à l'aide de vecteurs, puis comme avec la création d'objets simple nous ajoutons notre triangle au monde.

```
var points = [{x: 0, y: 0}, {x: 1, y: 0}, {x: 0, y: 2}];
for (var i = 0; i < points.length; i++) {
    var vec = new b2Vec2();
    vec.Set(points[i].x, points[i].y);
    points[i] = vec;
}
this.fixDef.shape = new b2PolygonShape;
this.fixDef.shape.SetAsArray(points, points.length);
this.bodyDef.position.x = 5;
this.bodyDef.position.y = 5;
this.world.CreateBody(this.bodyDef).CreateFixture(this.fixDef);
```

Pour des objets plus avancés, je vais commencer à utiliser une terminologie plus exacte, maintenant que nous avons couvert les bases de placement des objets sur l'écran et l'interaction les uns avec les autres. Il est important de comprendre les spécificités maintenant que nous avançons dans la complexité.

- ❖ Un "Body" (un corps) est le morceau rigide de l'objet.
- ❖ Un "Shape" (une forme) est un objet géométrique comme un polygone, un carré, un cercle etc..., cet objet doit être convexe.
- ❖ La "Fixture" est la partie qui lie une forme à son corps.

Maintenant que nous avons défini cela, il faut savoir qu'avec Box2D, un corps peut avoir plusieurs formes ! Cela veut donc dire que l'on peut construire des objets ayant des formes plus avancées : tel que dans cet exemple, une maison ! Testez par vous-même :

```
var entity = "4": {id: 4, x: 10, y: 10, polys: [
    [{x: -1, y: -1}, {x: 1, y: -1}, {x: 1, y: 1}, {x: -1, y: 1}], // box
    [{x: 1, y: -1.5}, {x: 2, y: 0}, {x: 1, y: 1.5}] // arrow
]};
for (var j = 0; j < entity.polys.length; j++) {
    var points = entity.polys[j];
    var vecs = [];
    for (var i = 0; i < points.length; i++) {
        var vec = new b2Vec2();
        vec.Set(points[i].x, points[i].y);
        vecs[i] = vec;
    }
    this.fixDef.shape = new b2PolygonShape;
    this.fixDef.shape.SetAsArray(vecs, vecs.length);
    body.CreateFixture(this.fixDef);
}
```

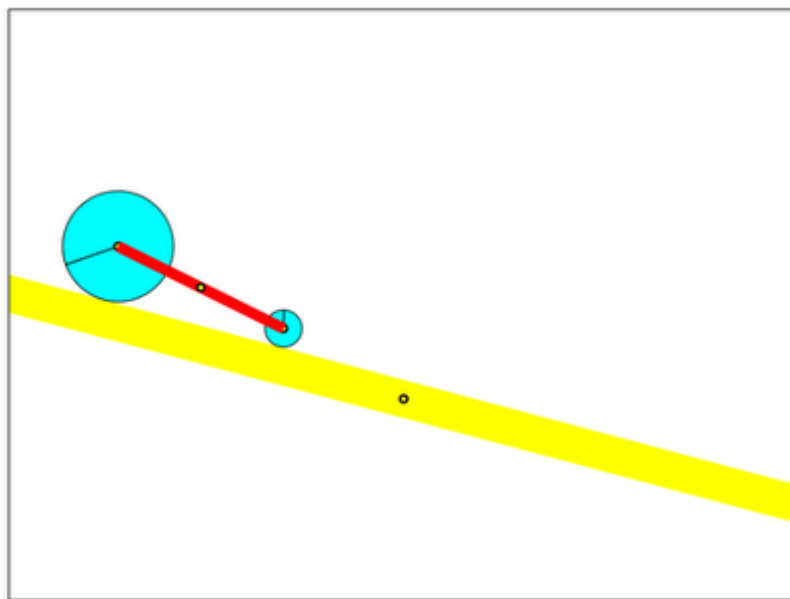
Vous pouvez retrouver cet exemple (ainsi que d'autres formes) à cette adresse : <http://box2d-javascript-fun.appspot.com/08/index.html> .

5. Articuler les objets entre eux

Auparavant, nous avons vu comment construire des objets concaves en associant deux ou plusieurs formes de corps entre elles. Une autre façon de construire des objets plus complexes consiste à lier entre eux à l'aide d'articulations « joints ».

De plus, les " joints " permettent de contraindre le mouvement de certains corps ensemble, par exemple en limitant l'amplitude des mouvements ou en affectant le mouvement d'un corps sur l'autre. Box2D prend en charge de nombreux types de " joints ", y compris les tirages, les engrenages et les cordes. Pour une liste approfondie des différents types ainsi que des exemples, référez-vous à ce manuel : http://www.box2d.org/manual.html#_Toc258082974.

Pour débiter, nous allons réaliser une "liaison pivot" (Revolute joint), une façon élégante de dire une articulation qui permet à deux corps pour tourner autour d'un point commun. Voici ce que nous allons réaliser :



Pour vous entraîner, créez par vous-même les deux cercles, si vous n'y arrivez pas référez-vous à la version finale de cet exemple ici : <http://box2d-javascript-fun.appspot.com/09/index.html> .

Créons maintenant notre "joint".

Comme la plupart des objets de Box2D, cela commence par une définition :

```
var joint = new b2RevoluteJointDef();
joint.Initialize(body1, body2, body1.GetWorldCenter());
this.world.CreateJoint(joint);
```

Il est donc très simple de définir un "joint", de plus les RevoluteJoint sont très facilement configurable. Voici un exemple des options que l'on peut leur appliquer :

- ❖ enableLimit - Si les limites communes sont actives.
- ❖ lowerAngle - Angle de la limite inférieure.
- ❖ upperAngle - Angle de la limite supérieure
- ❖ enableMotor - Si le moteur commun sera actif, par défaut il sera à false.
- ❖ MotorSpeed - La vitesse du moteur en commun. Positif pour le sens antihoraire, négatif dans le sens des aiguilles d'une montre.
- ❖ maxMotorTorque - Le couple maximal que le moteur peut utiliser. Un couple trop faible ne sera pas en mesure de déplacer les corps.

A l'aide de MotorSpeed et le maxMotorTorque nous pouvons faire une voiture auto-alimentée :

```
bTest.prototype.addRevoluteJoint = function(body1Id, body2Id, params) {
    var body1 = this.bodiesMap[body1Id];
    var body2 = this.bodiesMap[body2Id];
    var joint = new b2RevoluteJointDef();
    joint.Initialize(body1, body2, body1.GetWorldCenter());
    if (params && params.motorSpeed) {
        joint.motorSpeed = params.motorSpeed;
        joint.maxMotorTorque = params.maxMotorTorque;
        joint.enableMotor = true;
    }
    this.world.CreateJoint(joint);
}
```

Voici l'exemple en action : <http://box2d-javascript-fun.appspot.com/10/index.html>.

6. Appliquer des forces aux objets

Jusqu'à maintenant, nous avons vu comment les objets interagissent seulement avec la gravité, voire avec les articulations motorisées. Il est maintenant temps d'aller un peu plus loin en introduisant les impulsions. En utilisant les impulsions, on peut immédiatement changer la vitesse de nos objets.

Box2D a deux façons apparemment similaires d'injecter un peu de vie dans vos corps :

1. `ApplyImpulse` : Change immédiatement l'élan. C'est comme si on frappait l'objet.
2. `ApplyForce` : Change l'élan au cours du temps. C'est comme si on poussait l'objet.

Comment créer cette impulsion ?

Dans les deux cas elle a besoin d'un vecteur, d'où un `b2Vec2`, pour la direction et l'ampleur. Le reste du code reste facile, voici un moyen de créer facilement une impulsion :

```
bTest.prototype.applyImpulse = function (bodyId, degrees, power) {  
    var body = this.bodiesMap[bodyId];  
    body.ApplyImpulse (new b2Vec2 (Math.cos (degrees * (Math.PI / 180)) *  
power,                                     Math.sin (degrees * (Math.PI / 180)) *  
power) ,                                     body.GetWorldCenter() );  
}
```

Pour créer le vecteur de l'impulsion, la direction (en degrés) et la puissance (une valeur arbitraire appliquée ici pour tester les différentes puissances) sont combinées.

L'impulsion est lancée depuis le centre de masse du cercle. L'impulsion peut être lancée depuis n'importe quel point de l'objet ce qui crée une rotation sur l'objet.

Vous pouvez retrouver un exemple détaillé ici :

<http://box2d-javascript-fun.appspot.com/08/index.html> .

7. Gérer les collisions

Maintenant que nous avons réussi à donner une impulsion à nos objets, il faut pouvoir détecter leurs collisions.

Pour l'instant nous avons réagi à aucun évènement créé par Box2D, heureusement il existe un moyen simple pour être prévenu lorsque deux corps se touchent, quand ils arrêtent de se contacter et même combien d'impulsions se sont fait ressentir par les organismes. Nous allons voir tout cela.

La classe `b2ContactListener` de Box2D fournit ces quatre rappels que vous pouvez utiliser pour être informé de contacts connexes. Ces événements sont les suivants :

- `BeginContact` – déclenché lorsque deux corps commencent à se toucher.
- `EndContact` - déclenché lorsque deux corps ont arrêté de se toucher.
- `PreSolve` - déclenché avant que le contact est résolu. Vous avez la possibilité d'annuler le contact ici
- `PostSolve` - déclenché lorsque le contact est résolu. L'évènement comprend aussi l'impulsion du contact.

Attention, notez que tous ces événements sont déclenchés lors d'une étape d'avancement du monde. Cela implique donc qu'il faut être très prudent de ne pas manipuler le monde lors de ces évènements comme la simulation Box2D n'est pas terminée.

`EndContact` peut être déclenché à l'extérieur de l'étape dans le cas où l'objet est retiré du monde.

Voici un exemple simple de listener :

```
var listener = new Box2D.Dynamics.b2ContactListener;
listener.BeginContact = function(contact) {
    // console.log(contact.GetFixtureA().GetBody().GetUserData());
}
listener.EndContact = function(contact) {
    // console.log(contact.GetFixtureA().GetBody().GetUserData());
}
listener.PostSolve = function(contact, impulse) {
}
listener.PreSolve = function(contact, oldManifold) {
}
this.world.SetContactListener(listener);
```

Chacun de ces évènements prend en paramètre le contact qui contient les détails de ce contact, les paramètres importants de ce contact sont `GetFixtureA()` et `GetFixtureB()` qui retournent les "fixture" intervenants dans le contact. C'est depuis ces "fixtures" que vous pouvez référencer les corps.

En effet, attention, les collisions se font entre "fixtures" et non entre les corps.

Maintenant que l'on connaît les objets qui sont rentrés en contact, il serait intéressant de savoir comment a été l'impact, de quelle puissance ?

En regardant de près, vous remarquerez que Box2D ne peut pas vous remettre la force ou l'impulsion à l'intérieur BeginContact(). Vous pouvez bien sûr calculer votre propre valeur pour l'amplitude de la collision, en utilisant linearVelocity et mass sur la "fixture" trouvée.

Cependant PostSolve() inclut un objet "impulse", qui comprend un tableau de valeurs d'impulsion de la collision. Cette valeur est la meilleure façon d'accéder à l'ampleur de la collision.

Si seulement les choses étaient si faciles !

PostSolve() est appelée à chaque fois qu'une "fixture" d'un corps sent une impulsion d'un autre organisme. Si l'objet A frappe l'objet B qui est relié à l'objet C, d'où l'objet C ressent l'impulsion initiale de l'objet A. Un coup de départ (parce que l'impulsion a frappé l'objet B qui a "frappé" l'objet C). Cela peut ou peut ne pas être ce que vous voulez, selon votre logique de jeu.

Par exemple, si vous avez une balle qui roule sur le sol, ce qui crée une impulsion (bien que petite) pour chaque trame, et PostSolve() sera déclenché à chaque image. Si vous avez d'autres objets touchant le sol, ces objets ressentiront une impulsion à partir du sol.

Si vous avez l'intention de calculer l'endommagement d'un objet à partir d'un impact, il faut être très conscient que PostSolve() sera déclenché de nombreuses fois. D'où si, par exemple, vous souhaitez soustraire un total des dommages à votre objet ayant un certain nombre de points de vie à chaque appel de PostSolve() alors il faut être conscient que votre objet subira les dommages causés par pratiquement chaque impulsion.

Vous pouvez bien sûr créer des "points de vie" initiaux assez élevées pour résister à des impulsions de partout, ou de rejeter des impulsions inférieures à un certain seuil. Vos besoins peuvent varier.

Le point important à retenir est que PostSolve() sera appelée un grand nombre de fois.

Pour aller plus loin sur les collisions, vous pouvez aller voir cet exemple ainsi que son explication : <http://box2d-javascript-fun.appspot.com/12/index.html>.

8. Aller plus loin

Si vous souhaitez approfondir votre connaissance sur Box2D, vous pouvez consulter ces différents sites web (Attention, toutes ces pages sont en anglais) :

- Pour voir d'autres tutoriels :
 - ❖ <http://aniruddhaloya.blogspot.fr/2012/11/box2d-javascript-part-2.html>, un tutoriel expliquant les bases de Box2D en JavaScript.
 - ❖ <http://creativejs.com/2011/09/box2d-javascript-tutorial-series-by-seth-ladd/>, un site regroupant les différents liens vers les articles du blog de [Seth Ladd](#) qui a réalisé un excellent tutoriel sur Box2D en JavaScript. Ainsi qu'une version simplifiée en diaporama : <http://ongamestart-2011-box2d.appspot.com/#1>.
- Pour se documenter sur Box2D :
 - ❖ <http://www.box2dflash.org/docs/2.1a/reference/>, l'API Flash de Box2D qui a été reprise afin de réaliser Box2D en JavaScript, on retrouve donc toutes les classes nécessaires.
 - ❖ <http://www.box2d.org/manual.html>, le manuel de Box2D expliquant toutes les fonctionnalités réalisables à partir de celui-ci.