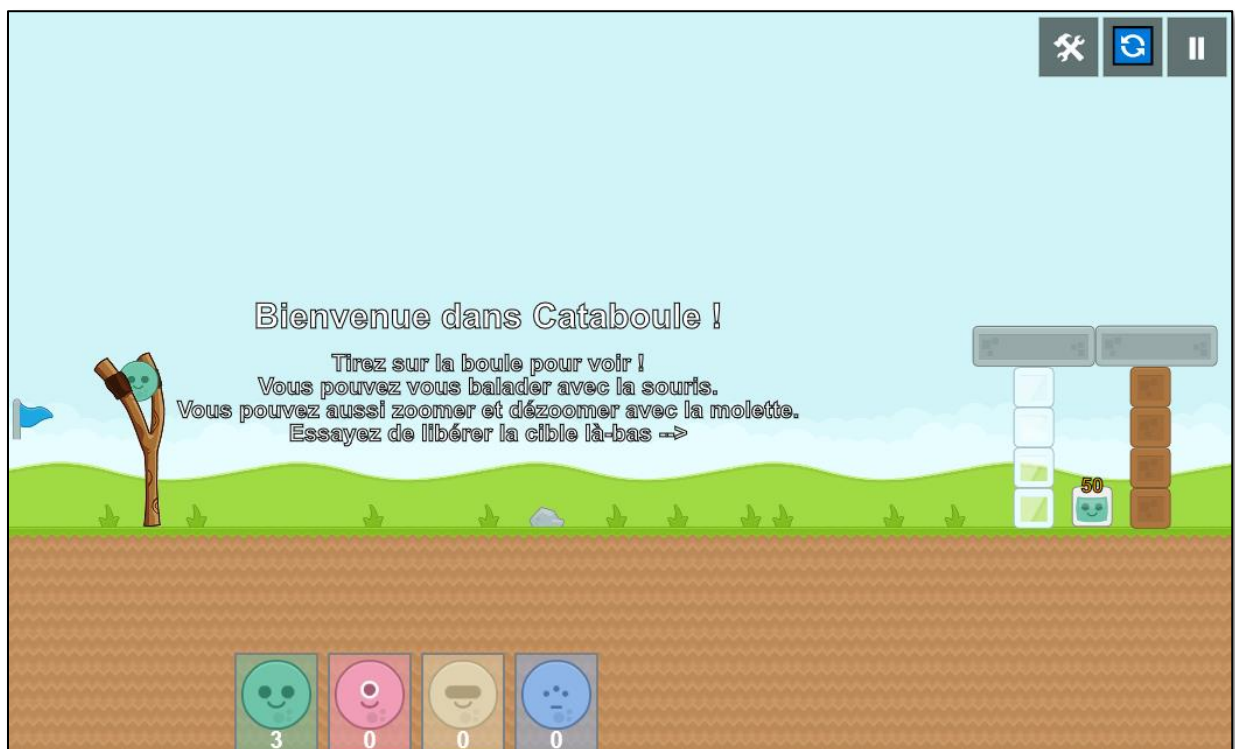


Alexandre l'Heritier

# Rapport

Projet de Programmation JavaScript : Projet Cataboule

GitHub : <https://github.com/AlexlHer/Cataboule>



## Table des matières

Introduction.....	- 2 -
Modification du code TP2.....	- 2 -
Organisation du code.....	- 2 -
main.js.....	- 2 -
Engine/Canvas .....	- 3 -
Jeu.....	- 3 -
elemsInterface .....	- 4 -
Rect/Body.....	- 4 -
Rond/Carre .....	- 4 -
Boule/Case/Cible/Mur .....	- 4 -
Exemples de code complexe que j'aurai pu écrire .....	- 5 -
Spécification du format de niveaux.....	- 5 -
JSON .....	- 5 -
Eléments.....	- 8 -
Case.....	- 8 -
Boule/Cible .....	- 9 -
Background .....	- 10 -
Décor .....	- 10 -
Murs.....	- 10 -

## Introduction

Le code du projet Cataboule a été écrit en intégralité par moi, mis à part le code du TP2. Au début, j'ai tenté d'écrire un moteur physique plus complet mais j'ai abandonné au vu de l'ampleur de la tâche. Cela étant, j'ai pris le code du TP2 comme point de départ du projet.

J'ai utilisé la documentation de Mozilla (<https://developer.mozilla.org/fr/>) ou la documentation de W3Schools (<https://www.w3schools.com/jsref/>) (la première qui vient sur Google lors d'une recherche) et lorsque je suis bloqué, <https://stackoverflow.com>.

Le projet implémente toutes les fonctionnalités de base et toutes les fonctionnalités avancées. Petites précisions sur quelques fonctionnalités avancées :

- « Objet friable » et « Blessures » s'applique uniquement si la collision est faite avec une certaine vitesse ( $> 0.4$ ).
- « Jeu complet » la logique de jeu est de détruire les cibles avec les boules disponibles (qui sont limitées). Pas de défaite : si le joueur n'arrive pas à détruire toutes les cibles, pour avoir accès au prochain niveau, il doit recommencer.
- Il y a un mode debug qui peut être pratique pour tester le jeu (« Constants.debugMode = true » dans la console)

## Modification du code TP2

Lors de la build B200218.1, j'ai remplacé toute la gestion basée sur le déplacement des <div> par du déplacement de rectangles dessinés sur canvas.

J'ai fusionné les classes Renderer et Engine puis j'ai créé la classe Canvas pour gérer le canvas. J'ai découpé la classe Sprite en deux : Carre et Rond (qui est un carré mais avec un dessin de rond). J'ai ajouté une origine initialisée à width/2, height/2 sur laquelle tous les futurs objets devront prendre comme origine. Cela m'a permis de créer différentes méthodes permettant de déplacer l'origine (et donc tous les objets avec des positions relative à celle-ci).

Comme cette origine est mathématique (y vers le haut), j'ai inversé la gravité puis j'ai ajouté un zoom avec la molette. Enfin, j'ai ajouté une boule pouvant être lancé avec la souris.

Finalement, j'ai conservé les principales parties du code : tout ce qui est collisions, vector, rect.

## Organisation du code

main.js

- Crée Engine et Jeu et les lie pour qu'il puisse se partager des infos.

- S'occupe de l'événement `resize` : appel de `engine.set_size_canvas()` lors d'un redimensionnement de la fenêtre.
- Contient la boucle principale qui appelle `engine.update()`.
- Compte les FPS.
- Ajoute les événements liés à la souris.

## Engine/Canvas

Engine (`engine.js`) et Canvas (`canvas.js`) s'occupent du canvas

- Ils gèrent tous les Body du jeu (boules, cases, murs, cibles) :
  - o Ajout/Suppression.
  - o Déplacements.
  - o Appels des `draw()` des Body
  - o Lancement des boules
  - o Pouvoir des boules
- Dessin du background :
  - o Images de background.
  - o Textes dans le jeu (les instructions pour le joueur par exemple, enregistrées dans les sauvegardes).
  - o Décors dans le jeu (herbes, cactus, drapeaux).
- Déplacements/événements dans le canvas :
  - o Cliques et déplacements de la souris.
  - o Zoom et dézoom avec la molette.

## Jeu

Jeu (`jeu.js`) s'occupe de l'organisation du jeu

- Il gère toutes les interfaces :
  - o Interfaces « plein écran », qui prennent tout l'écran et qui prennent la main sur le canvas (Engine transmet les cliques et les déplacements de la souris, Jeu dessine directement grâce au contexte de Canvas ; la liaison des deux sert à ça).
  - o Interface « en jeu », qui s'affiche lors du jeu. Cette fois, à la fin de chaque `Engine::update()`, Engine appelle `Jeu::update()` qui dessine l'interface sur le jeu. À chaque clique, Engine demande à Jeu si le clique est sur un élément de l'interface. Si oui, Engine passe la main à Jeu, sinon Engine continue. Engine envoie chaque déplacement de souris à Jeu.
  - o Fond pour les interfaces « plein écran », crée dynamiquement selon la taille du canvas.
- Il gère le déroulement du jeu :
  - o La sauvegarde et la restauration de la progression dans les cookies.
  - o La récupération et la lecture des niveaux `.json`.
  - o Le changement de boule.

- La victoire du joueur.

### elemsInterface

Les interfaces sont constituées de trois choses : Bouton, Texte et Imagee (elemsInterface.js)

- Bouton représente un bouton sur lequel on peut cliquer dessus. Bouton s'occupe de l'affichage du bouton : Rect avec une couleur de background et une couleur de contour. Lorsque la souris passe dessus, il y a un changement de couleur (on donne la position de la souris lors du dessin du bouton). Ce Bouton crée aussi un Texte qui s'affichera sur le bouton.
- Texte représente un texte qui n'est pas cliquable. Il a une couleur pouvant être personnalisée et un contour noir avec une épaisseur personnalisable.
- Imagee représente simplement une image, avec une position et une taille.

### Rect/Body

Rect n'a pas changé par rapport au TP2, c'est toujours une classe représentant un rectangle avec une position et une taille et a toujours trois méthodes (une pour déplacer le Rect, une autre pour faire la différence avec un autre Rect et une dernière pour savoir si l'origine est dans le Rect).

Body a, en revanche, été modifié avec notamment l'ajout de niveaux de vie, la possibilité d'un déplacement défini et infini. C'est collision() qui gère les niveaux de vie.

### Rond/Carre

On a deux types de Body : Rond et Carre (sprite.js)

- Carre est simplement un Body avec une méthode draw() qui permet de dessiner le rectangle dans le canvas (via le ctx passé en argument au constructeur).
- Rond a la même signature que Carre (ce qui permet de les interchanger) mais est forcément carré (on garde la plus grande valeur entre width et height pour le diamètre) et on calcule un attribut rayon. Rond::draw() est presque identique à Carre::draw() mais dessine un arc au lieu d'un rect.

### Boule/Case/Cible/Mur

On a aussi plusieurs types de Rond et de Carre : Boule, Case, Cible, Mur

- Les quatre types redéfinissent draw() pour dessiner une image au lieu d'un arc ou d'un rect.
- Boule (Rond) a la caractéristique d'avoir une masse Infinity au début puis une masse définie. Boule a aussi un pouvoir qui peut être déclenché après

lancement. Boule0->3 définissent des pouvoirs différents et des taille/masse/élasticité différentes.

- Case est un Carre mais avec 3 images représentant les paliers de vie de la Case. Case1->8 représentent les formes de Case différentes (voir Doc plus bas). Une exception : Case0 qui est un Rond. Chaque case peut aussi avoir 4 types de matériaux différents, ce qui joue sur leur masse, leur élasticité et leur niveau de vie.
- Cible est un Carre avec une image et un type. Cible 0->3 sont juste 4 cibles avec taille et masse différentes.
- Mur est un Carre avec une élasticité pouvant être personnalisé. Le draw() est ici un peu plus fourni pour pouvoir recouvrir tout le mur d'images. MurInvisible est un Carre avec un draw() vide (sauf si debug actif).

Vu qu'un Rond est un Rect avec un draw() qui dessine un arc, on aurait pu extends juste Carre mais c'est en cas d'évolution des collisions avec Rond.

### Exemples de code complexe que j'aurai pu écrire

Lors de la première version du projet, j'ai essayé d'écrire un moteur physique plus complet que celui du TP2, avec des collisions plus réalistes (collisions entre carré et rond par exemple, avec des rotations et tout) mais, vu que j'ai mis une journée pour faire la collision entre le sol (fixe) et un rond puis entre deux ronds, j'ai arrêté pour pouvoir mettre l'accent sur le reste du projet.

J'ai d'abord cherché sur internet voir si je trouvais comment coder ça. Je suis tombé là-dessus : <https://gregorycorgie.developpez.com/tutoriels/physic/> . J'ai essayé de suivre la partie « Collision Sphère/Plan » puis la partie « Collision Sphère/Sphère » mais je n'ai pas tout compris. J'ai quand même tenté. La partie Rond/Sol fonctionnait bien mais la partie Rond/Rond faisait n'importe quoi. J'ai donc abandonné (peut-être pour un projet perso, plus tard) et j'ai décidé de me concentrer sur l'implémentation des fonctionnalités de base / avancées à partir du TP2.

### Spécification du format de niveaux

#### JSON

Pour qu'un niveau soit pris en compte dans le jeu, il doit se trouver dans le dossier « levels » et avoir comme nom « levelX.json » (avec X nombre entier >= 1)

Les X doivent se suivre, il ne doit pas y avoir de saut (level10.json sans level9.json par exemple).

Le niveau peut être accompagné d'une image se nommant « levelX.webp » (X identique à celui du niveau).

Il est aussi possible de charger un niveau stocké localement à partir du menu « Accueil » puis « Voir les niveaux ».

```
{
// -----
// Eléments obligatoires :
// -----

// Emplacement de la boule et du lanceur de boule (si boule à y = 0,
// pied du lanceur à y = -250).
// x, y désigne le point centre de la boule.
// [x, y]
"emplacement_boule": [-75, 250],

// Le nombre de boules disponible pour le niveau.
// [type 0, type 1, type 2, type 3]
"nb_boule": [1, 1, 1, 1],

// Le type et l'emplacement des cibles.
// x, y désigne le point haut gauche de la cible.
// [[type cible, x, y], ...]
"type_emplacement_cibles": [
    [0, 620, 165]
],

// Le type, la forme, l'emplacement et l'élasticité des cases.
// x, y désigne le point haut gauche. L'élasticité est facultative.
// [[type, forme, x, y, élasticité], ...]
"type_forme_emplacement_cases": [
    [0, 6, 500, 220],
    [1, 6, 740, 220, 1],
    [2, 3, 550, 295],
    [3, 2, 590, 435],
    [0, 1, 620, 85]
],

// Ici, seul un des deux types de mur est obligatoire
// (il peut y avoir les deux, mais au moins un est nécessaire).

// L'épaisseur et l'emplacement du cadre invisible.
// xy1 désigne le point haut gauche et xy2 le point bas droit du
// Cadre (les limites du jeu).
// [épaisseur, x1, y1, x2, y2]
"epaisseur_emplacement_murs_i":
[100, -1000, 1000, 4000, -70],

// Le type, l'emplacement, la taille et l'élasticité des murs visible
// x, y désigne le point haut gauche. L'élasticité est facultative.
// [[type, avec/sans herbe, x, y, width, height, élasticité], ...]
```

```

"type_emplacement_taille_murs_v": [
    [2, true, -500, 0, 3090, 120]
],
// -----
// Éléments facultatifs :
// -----

// Le type et l'emplacement de la déco.
// x, y désigne le point bas centre.
// [[type, x, y], ...]
"type_emplacement_deco": [
    [0, 0, 0], [1, 100, 0], [2, 200, 0], [3, 300, 0],
    [4, 400, 0], [5, 500, 0], [6, 600, 0], [7, 700, 0]
],

// Le type de background.
// [normal/bleu, apparence]
"type_bg": [0, 3],

// L'emplacement, la taille et le texte des textes.
// x, y désigne le point centre centre.
// [[x, y, taille, texte], ...]
"emplacement_taille_textes": [
    [0, 100, 40, "test"]
],

// Le type, la forme, l'emplacement A, l'emplacement B,
// la vitesse de déplacement et l'élasticité des cases mouvantes.
// La case ira de A vers B puis de B vers A, &c.
// x, y désigne le point haut gauche. L'élasticité est facultative.
// [[type, forme, xA, yA, xB, yB, vitesse, élasticité], ...]
"type_forme_emplacement1_emplacement2_vitesse_cases_m": [
    [0, 1, 100, 400, 600, 200, 2]
],

// Le type, l'emplacement A, l'emplacement B, la taille
// la vitesse de déplacement et l'élasticité des murs mouvants.
// La case ira de A vers B puis de B vers A, &c.
// x, y désigne le point haut gauche. L'élasticité est facultative.
// [[type, xA, yA, xB, yB, width, height, vitesse, élasticité], ...]
"type_emplacement1_emplacement2_taille_vitesse_murs_m": [
    [2, true, 200, 400, 700, 200, 60, 60, 2]
],

// Si on veut un niveau sans gravité.
"zeroGravity": true,

// La force du lanceur (1 par défaut).
"forceLanceur": 1,

```

















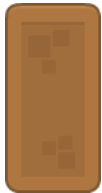

```
// Le niveau de zoom initial (0.8 par défaut).
"zoomInitial": 0.5
}
```

## Éléments

### Case









Il y a 108 cases qui peuvent être utilisées. Dans le JSON, on demande le type et la forme (l'état est géré par le jeu) et la taille est toujours en px :

Type	Forme	Etat
0 Glass : 	0  (1x1)ua / (70x70)px	0 
1 Wood : 	1  (1x1)ua / (70x70)px	1 
2 Stone : 	2  (2x2)ua / (140x140)px	2 
3 Metal : 	3  (3x1)ua / (210x70)px	
	4  (2x1)ua / (140x70)px	
	5  (3x2)ua / (210x140)px	

	 6 (1x3)ua / (70x210)px	
	 7 (1x2)ua / (70x140)px	
	 8 (2x3)ua / (140x210)px	

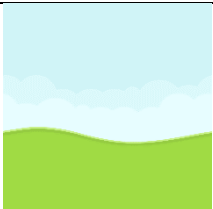
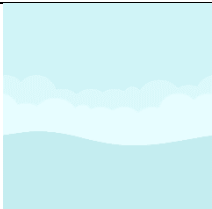


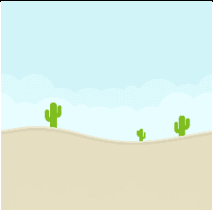
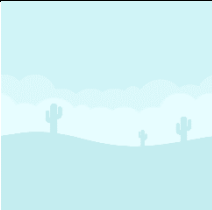
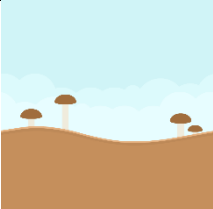

#### Boule/Cible

Il y a 4 boules et cibles différentes. Lorsqu'une cible est « libérée », elle fait son pouvoir et se dirige vers une autre cible (la cible n°0 dans la liste interne des cibles).

Boule	Cible		Boule	Cible
0  Sans pouvoir	0 		2  Pouvoir de vitesse	2 
1  Pouvoir de grossissement	1 		3  Pouvoir de triplcation	3 









## Background

Il y a 2 types de backgrounds : coloré et bleu. Et dans ces 2 types, il y a 4 ambiances.

0. Coloré		1. Bleu	
0		0	
1		1	
2		2	
3		3	

## Décor

Il y a 8 décors différents.

0	1	2	3	4	5	6	7
							

## Murs

Il y a 5 types de murs avec un dessus et un dessous.

0	1	2	3	4
