

# Phaser

## Jeux vidéo 2D en Javascript

Rudi Giot - 9 février 2018



**Attribution - Pas d'Utilisation  
Commerciale - Pas de Modification 3.0  
non transposé (CC BY-NC-ND 3.0)**

---

1. Introduction	5
1.1.HTML5 et Javascript	5
1.2.Phaser	5
1.3.Serveur HTTP	6
1.4.Sandbox	8
1.5.Installation de Phaser	8
1.6.Exercice « fil rouge »	9
2. Les bases	10
2.1.Structure d'un programme	10
2.2.Machine à états	11
2.3.Variables	13
2.4.Commentaires	14
2.5.Debug	14
2.6.Boucles	15
2.7.Physique	16
2.8.Saisie clavier	17
2.9.Instructions conditionnelles	17
2.10.Fonctions	20
2.11.Tableaux	20
3. Construction de maps	22
3.1.Groupe d'objets	23
3.2.TileSet	24
3.3.JSON	26
4. Evénements temporels	29
5. Physique	31
5.1.Vecteurs	31
5.2.Forces	33
5.3.Collisions	33
6. Animations	34

---

6.1.Sprite animé	34
7. Classes et objets	35
7.1.Définition d'une classe	35
7.2.Propriétés	35
7.3.Méthodes	36
7.4.Groupe d'objets	37
7.5.Gestion des groupes d'objets	37
8. Camera	38
8.1.Scrolling	38
9. Audio	39
10.Sauvegarde de données	40
11.Intégration dans une page HTML	41
12.Organisation du code	42
12.1.Machine à états finis	42
12.2.Segmentations du code en plusieurs fichiers	45

---

# Préface

Ce document est destiné aux programmeurs qui désirent apprendre *Phaser* avec comme but de développer des applications 2D interactives en *Javascript*. *Phaser* est typiquement utilisé pour développer des jeux vidéo 2D qui tournent dans un *browser*. Ce cours **n'est pas destiné** aux débutants en programmation. Si vous n'en connaissez pas les éléments de base (variables, conditions, boucles, ...), référez vous à d'autres documents avant d'aborder celui-ci. Ce syllabus nécessite également quelques connaissances élémentaires en *HTML* et en *CSS*.

Il y a deux types d'exercices dans le syllabus, ceux qui sont obligatoires pour la bonne compréhension de la suite du cours (la solution est donnée dans des fichiers annexes) et ceux qui sont facultatifs (complémentaires) et qui ne sont pas résolus.

Tous les éléments de ce document sont originaux sauf certains passages ou illustrations qui sont alors référencés. Ce document est mis à disposition sous licence Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 non transposé. Pour voir une copie de cette licence, visitez <http://creativecommons.org/licenses/by-nc-nd/3.0/>.

Toutes les solutions des exercices ainsi que les éléments graphiques et sonores (assets) sont disponibles sur GitHub à l'adresse :

<https://github.com/RudiGiot/Phaser/>

Bonne lecture et surtout bon amusement.

# 1. Introduction

## 1.1.HTML5 et Javascript

L'*HTML5* (*HyperText Markup Language* - version 5) est la version actuelle (spécifications officielles en 2014) de l'*HTML*, le langage de « marquage hyper-texte ». Dans le langage courant, l'*HTML5* désigne un ensemble de technologies destinées aux navigateurs *Web* et inclut les notions d'*HTML*, de *CSS* et de *JavaScript*.

Le *JavaScript* est un langage de programmation « orienté objet à prototype » créé en 1995. Sa syntaxe se base sur celle du langage *JAVA* mais la ressemblance s'arrête là. Leur utilisation et leur fonctionnement sont complètement différents, ne faites jamais l'amalgame entre les deux. Le *JavaScript* est traditionnellement utilisé au sein de pages *HTML*, dans des navigateurs *Web* mais est également exploité dans des technologies côté serveur, comme dans *Node.js*, par exemple. Il existe de nombreux « *framework* » et librairies qui facilitent la programmation en *JavaScript*. *JQuery* et *Three.js* sont sans doute les plus connues.



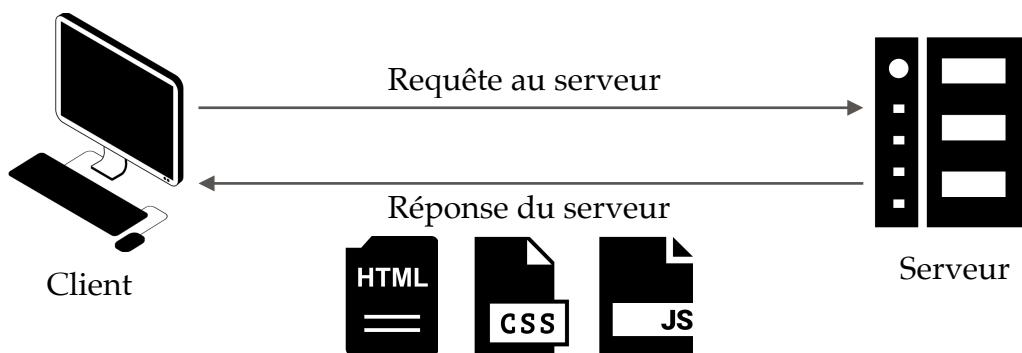
*Exemples d'applications réalisées avec Phaser, Node et Three*

## 1.2.Phaser

*Phaser* est basé sur un ancien *framework* appelé *Flixel* qui était développé pour l'*ActionScript*. *Phaser* est actuellement développé par *Richard Davey*. Les versions se suivent régulièrement, sont améliorées, optimisées et « débuguées ». De plus, le site <http://phaser.io> comporte de nombreux exemples, tutoriels et un forum très actifs. On peut programmer un projet *Phaser* avec les langages *TypeScript* et *JavaScript*. *Phaser* est gratuit et *open-source*, il fait donc un excellent candidat pour l'apprentissage de la programmation de jeux vidéos ou d'application graphique interactives en 2D destinés à des navigateurs *Web* (compatibles *HTML5*).

## 1.3.Serveur HTTP

Pour pouvoir travailler en *JavaScript* avec *Phaser* nous avons besoin d'un « serveur Web ». En effet, la politique de sécurité des *Browsers* interdit aux *Scripts* l'accès au système de fichier local. Il faut donc obligatoirement déposer ses fichiers (sources, images, sons, ...) sur un serveur *HTTP*. Ce sigle est l'abréviation de « *HyperText Transfer Protocol* », un protocole qui permet le transfert de fichiers. Son fonctionnement est relativement simple: un client *HTTP* (*Firefox*, *Edge*, *Chrome*, ...) va contacter un serveur *HTTP* (*IIS*, *Apache*, ...) pour lui demander (requête *HTTP*) un fichier (souvent une page *HTML* ou des images), le serveur va lui répondre (réponse *HTTP*) avec un code (200, 404, ...) et le fichier s'il est disponible.



*Schéma d'une communication utilisant l'HTTP*

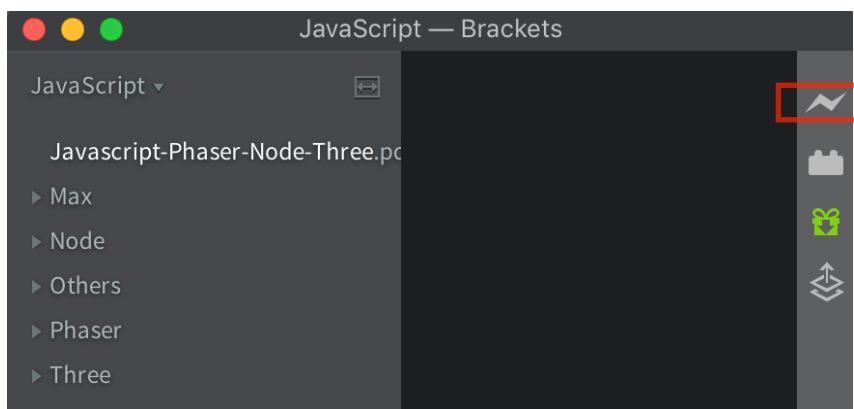
L'installation d'un serveur Web « opérationnel » (exposé sur *Internet*) est une opération très complexe. En effet, sa sécurisation est primordiale et nécessite des connaissances approfondies dans le domaine des réseaux. Heureusement, dans notre cas, nous n'avons besoin que d'un serveur *HTTP* de développement, un service qui s'exécute localement sur notre machine et qui ne nécessite donc aucune précaution particulière en terme de sécurité. Il y a deux possibilités qui s'offre à vous. Soit vous installez un vrai serveur *HTTP* (style *Apache*) ou alors vous utilisez la fonctionnalité de visualisation rapide d'un éditeur de pages *HTML* (*Brackets*, par exemple). Nous allons voir les deux méthodes, vous choisirez celle qui vous paraît la plus pratique.

### 1.3.1. \_AMP

Nous pouvons utiliser une distribution libre et gratuite du serveur *Apache* qui est intégré dans *WAMP*<sup>1</sup> (pour *Windows*), *MAMP*<sup>2</sup> (pour *MacOS*) ou *LAMP*<sup>3</sup> (pour *Linux*). Référez-vous aux sites cités en note de bas de page pour l'installation d'un de ces logiciels, celui qui correspond à votre système d'exploitation.

### 1.3.2. Brackets

Le logiciel *Brackets* est un éditeur de texte gratuit très utilisé pour l'*HTML* et le *CSS* mais également pour le *JavaScript*. Il permet de visualiser les pages en lançant un serveur *HTTP* minimal qui permet de visualiser ses pages en temps réel. Cette fonctionnalité est très utile dans notre cas puisqu'elle permettra d'exécuter notre code et de visualiser les images, écouter le son sans se soucier des problèmes de sécurité lié à l'exécution des scripts et de l'accès aux ressources locales. *Brackets* va donc se comporter comme un serveur *HTTP* pour notre *Browser*. Il vous suffit donc d'installer *Brackets*, de taper votre code, de le sauvegarder et de visualiser le résultat en cliquant sur le « petit éclair » en haut à droite de l'écran.



*Utiliser Brackets comme serveur HTTP*

#### Exercices :

- Visualisez dans votre navigateur *Internet* la page d'accueil de votre serveur local (<http://localhost/>) après avoir installé *\_AMP* ou *Brackets*.
- Créez un répertoire qui contiendra les exercices et projets que nous réaliserons par la suite

<sup>1</sup> <http://www.wampserver.com>

<sup>2</sup> <https://www.mamp.info>

<sup>3</sup> <https://fr.wikipedia.org/wiki/LAMP>

## 1.4.Sandbox

Il existe aussi la possibilité de travailler dans un *sandbox*, c'est à dire un environnement dans lequel vous pouvez tester rapidement vos programmes. Cette solution est plus simple pour débuter mais montre ses limites à un moment donné. Elle est idéale pour tester rapidement des portions de code ou développer un petit prototype rapidement. Ce « bac à sable » est disponible à l'adresse : <http://labs.phaser.io/edit.html>.

## 1.5.Installation de Phaser

Une fois le serveur *HTTP* installé et testé, nous allons procéder à l'installation de *Phaser* qui est une opération relativement simple. Il suffit, en effet, de télécharger le « *framework* » sur le site [phaser.io](http://phaser.io) et de copier le « *package* » complet dans le répertoire racine de votre serveur *HTTP* ou votre répertoire de travail dans *Brackets*. Ensuite, vous pouvez visualiser dans votre *browser* la page *HTML* qui se trouve dans le répertoire :

```
.../phaser-x.y.z/resources/tutorials/01 Getting Started/hellophaser/
```

Attention : remplacer les x, y et z par les numéros de la version installée.

Vous allez alors faire apparaître :



*Ecran d'accueil quand Phaser est correctement installé*

---

Si vous ne voyez pas cette image dans votre *browser* après avoir tapé l'*url* du dessus, vous avez sans doute mal copié les fichiers. Re-vérifiez la procédure.

## 1.6.Exercice « fil rouge »

Nous allons créer tout au long de ce cours un « rétro game », basé sur un ancien jeu qui date de la fin des années 80 : *R-Type*. Il s'agit d'un jeu de tir à scrolling horizontal. Nous allons le construire, pas à pas au fur et à mesure des nouvelles notions que nous aborderons.



*Screenshot du R-Type original<sup>4</sup>*

---

<sup>4</sup> <http://scoop.previewsworld.com/Home/4/1/73/1016?articleID=198525>

## 2. Les bases

### 2.1. Structure d'un programme

Une application *Phaser* vient s'intégrer dans une page *HTML*. On va donc retrouver la structure habituelle d'une page *HTML* avec l'entête *<head>* et le corps *<body>* :

```
<html>
  <head>
    <title>Exercice 1</title>
    <script src="../phaser-x.y.z/build/phaser.min.js"></script>
  </head>
  <body>
    <script type="text/javascript">

      // Votre code ici

    </script>
  </body>
</html>
```

La première balise *<script>* permet de faire une référence au *framework Phaser*. Assurez-vous que le chemin (*path*) est le bon et remplacez *x*, *y* et *z* par la valeur de votre version. A l'intérieur de la seconde balise *<script>*, on va retrouver le code de l'application qui commence de la manière suivante :

```
window.onload = function() {

  var game = new Phaser.Game(800, 600, Phaser.AUTO, '',
                            {create: create});

  function create ()
  {
    // Votre code
  }
};
```

On va ensuite pouvoir remplir la fonction *create()* avec des déclarations et des instructions.

---

Par exemple :

```
alert("Hello world");
```

La fonction *JavaScript alert("message")* permet d'afficher un « message » dans une fenêtre *pop-up*. On utilisera parfois cette fonction pour débogguer notre code.

Exercice :

- Assemblez les différentes portions de code précédentes dans un seul fichier (*.html*) et testez-le dans votre browser. Solution dans le fichier *Alert.html*.

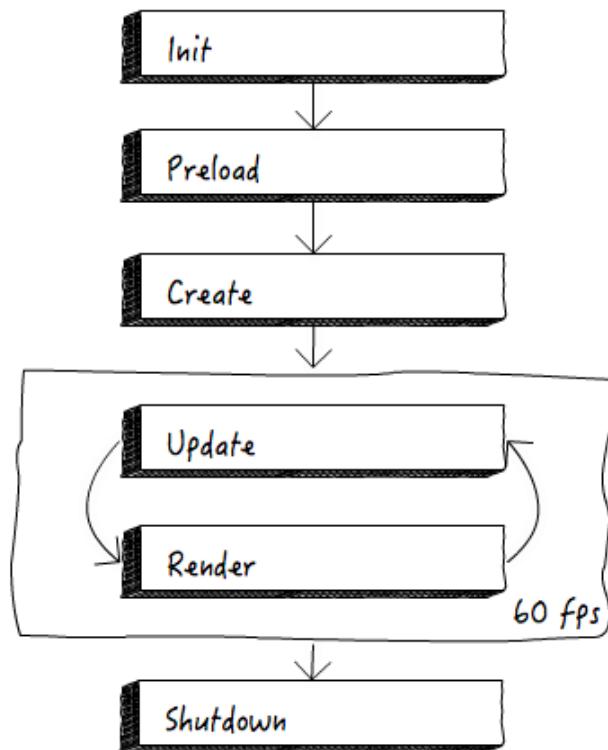
## 2.2.Machine à états

*Phaser* est exécuté à la manière d'une « *state machine* », c'est à dire une série de fonctions (leur nom est donc réservé) qui sont exécutées dans un ordre prédéfini. Ces fonctions doivent contenir votre code qui sera alors exécuté au moment de l'appel de cet état de la *state machine*. Les principales fonctions sont :

- *init()* : est la toute première fonction appelée lorsque l'application démarre à la création de l'objet *Phaser.Game()*. Elle est surtout utilisée pour préparer un ensemble de variables et / ou d'objets avant le pré-chargement des *assets* (images, son, vidéo, ...).
- *preload()* : est la fonction qui contient généralement le code qui charge les assets dont on aura besoin dans le jeu. Lorsque cette fonction est appelée les fonctions *update()* et *render()* ne sont pas encore exécutées. Si vous avez besoin d'une ou des deux fonctions sachez qu'il existe *loadUpdate()* et *loadRender()*, mais nous ne les utiliserons pas dans ce cours.
- *create()* : est la fonction qui est appelée automatiquement quand le pré-chargement est terminé. C'est ici que l'on crée généralement les *sprites*, les particules, les décors et tout ce dont on aura besoin comme objets durant l'exécution de l'application.
- *update()* : est la fonction appelée en boucle à chaque re-calculation de *frame*. L'appel à cette fonction est donc rythmé en fonction des performances de l'application et de la machine sur laquelle elle tourne. Le *frame rate* tourne généralement autour des 60. C'est donc dans l'*update()* qu'on déplacera les sprites, la caméra, qu'on détectera les collisions, c'est le cœur de votre application.

• *render()* : est la fonction appelée après que le rendu *WebGL/canvas* aie été calculé. Pratiquement on utilisera cette fonction pour des effets de post-rendu ou pour placer une couche visuelle de débogage.

• *shutdown()* : est la fonction appelée lorsque vous changer d'état dans votre jeu. Nous verrons son utilité tout à la fin de ce document.



*Machine à état de Phaser<sup>5</sup>*

<sup>5</sup> <https://mozdevs.github.io/html5-games-workshop/en/guides/platformer/the-game-loop/>

## 2.3.Variables

Le nom d'une variable commence toujours par une lettre (minuscule ou majuscule) ou un caractère « \$ » ou « \_ ». Les caractères suivants ce premier symbole peuvent être des chiffres. Le « *typage* » en *Javascript* est dit « *faible* ». Cela signifie que l'on peut déclarer une variable simplement en utilisant « *var* ».

Attention : *JavaScript* est « *case-sensitive* », vous devez donc toujours veiller à bien respecter les minuscules/majuscules. L'exemple suivant se trouve dans le fichier « *exercice1-1.html* ».

```
var helloString = "Hello world";
var number = 5;

alert(helloString);
alert(number);
alert (helloString + ' ' + number);

number = "and friends";

alert(helloString + number);
```

### Remarques :

- On peut changer le type des variables en cours de programme par simple réaffectation.
- On peut concaténer des chaînes de caractères en utilisant l'opérateur « + ».

## 2.4.Commentaires

Comme dans beaucoup d'autres langages les doubles « *slashes* » ( / / ) servent à écrire des commentaires sur une ligne. Pour commenter plusieurs lignes de suite on utilise le « *slash-étoile* » ( /\* ) pour commencer et le « *étoile-slash* » pour terminer ( \*/ ).

```
// commentaire sur une seule ligne ... jusqu'à la fin de la ligne  
  
/*  
Un commentaire sur plusieurs lignes commence avec /*  
et continue  
jusqu'à ... */
```

## 2.5.Debug

Pour débogguer un code *JavaScript*, il est préférable d'utiliser *Chrome* comme navigateur. En effet, ses « options pour développeurs » sont nombreuses et très pratiques (console, elements, timeline, ...). Il est également toujours préconisé de l'ouvrir en « mode privé » de manière à éviter que le « cache » n'empêche le rafraîchissement complet (images, sons, ...) de votre application. Vous veillerez aussi à chaque fois qu'une modification de code ne produit pas l'effet escompté dans votre navigateur de vérifier que le fichier visualisé est bien celui que vous éditez.

Vous pouvez également utiliser des instructions :

```
alert("Variable i : " + i);
```

Pour réaliser un *breakpoint* dans un programme et afficher la valeur d'une variable.

Nous verrons aussi plus loin comment utiliser la fonction *render()* pour afficher l'état de variables lors de l'exécution de l'application.

Exercice :

- Utilisez les commentaires, les variables, la concaténation dans un programme récapitulatif qui teste ces différents notions et testez-le dans votre browser. Exemple de solution dans le fichier *VariablesAlert.html*.

## 2.6.Boucles

### 2.6.1. Boucle « for »

La boucle « *for* » a la syntaxe suivante :

```
for (var i=0; i<10; i++) {  
    ...  
}
```

Pour l'illustrer, nous allons, grâce à *phaser*, charger une image et l'afficher une dizaine de fois.

```
window.onload = function() {  
  
    var game = new Phaser.Game(800, 600, Phaser.AUTO, '',  
        {preload: preload, create: create});  
  
    function preload() {  
        game.load.image('etoile', 'star.png');  
    }  
  
    function create() {  
        for (var i=0; i<10; i++) {  
            game.add.sprite(20 + i * 50, 20 + i * 50, 'etoile');  
        }  
    }  
};
```

#### Exercices :

- En partant de l'exemple ci-dessus et en utilisant les fonctions *JavaScript* *Math.round()* et *Math.random()* qui respectivement arrondissent un nombre réel en entier et génère un nombre aléatoire, vous pouvez afficher une vingtaine d'étoiles à l'écran de manière aléatoire. La solution est dans le fichier *RandomStar.html*.
- A partir de l'exercice précédent, vous affichez une vingtaine d'étoiles (une par ligne horizontale) de manière aléatoire (sur la ligne). La solution est dans le fichier *RandomStarInLine.html*.

## 2.6.2. Boucle while()

En Javascript, la syntaxe de la boucle *while()* est la suivante :

```
while (condition) {  
    //code to execute ...  
}
```

### Exercice :

- Dans un des deux exercices précédent remplacez la boucle *for()* par une boucle *while()*. La solution est dans le fichier *RandomStarWhile.html*.

## 2.7.Physique

L'intérêt d'utiliser un *framework* comme *Phaser* est qu'il contient une série de fonctionnalités qui simplifient l'implémentation de théories complexes, par exemple, la physique. Imaginez que vous soyez obligé de calculer la trajectoire d'un projectile à chaque *frame*. C'est faisable mais fastidieux. *Phaser* propose donc, par exemple de gérer la physique des objets pour vous. Nous allons, par exemple, afficher une étoile et en faire une étoile filante en lui donnant une vitesse. L'opération se réalise en deux instructions (en gras dans le code ci-dessous), il faut d'abord signaler à *Phaser* que les lois de la physique s'appliquent à l'étoile et ensuite lui demander d'appliquer une vitesse (selon l'axe des X et/ou des Y) de la valeur désirée :

```
window.onload = function() {  
    var game = new Phaser.Game(800, 600, Phaser.AUTO, '',  
        {preload: preload, create: create});  
  
    var star;  
  
    function preload() {  
        game.load.image('star', 'star.png');  
  
    }  
  
    function create() {  
        star = game.add.sprite(100, 100, 'star');  
        game.physics.arcade.enable(star);  
        star.body.velocity.x = 100; }  
}
```

### Exercice :

- Modifiez le code ci-dessus pour faire avancer l'étoiles dans les deux directions x et y. La solution est dans le fichier *FlyingStar.html*.

## 2.8.Saisie clavier

Pour rendre nos applications interactives, nous allons commencer par saisir les touches du clavier, pour, par exemple, faire avancer, monter ou descendre un vaisseau spatial. Pour ce faire, dans *Phaser*, on crée d'abord un objet :

```
cursors = game.input.keyboard.createCursorKeys();
```

Cet objet permet de savoir si une touche est enfoncée grâce à ses propriétés. Par exemple, la propriété booléenne (vraie ou fausse) :

```
cursors.left.isDown
```

Dans cet exemple, cette propriété (*isDown*) est « vraie » quand la touche « flèche gauche » est enfoncée. Pour traiter ces propriétés booléennes, nous avons évidemment besoin d'instructions conditionnelles.

## 2.9.Instructions conditionnelles

### 2.9.1. Condition « if ... else ... »

La syntaxe des instructions conditionnelles est la suivante :

```
if (condition) {  
    ...  
}  
else {  
    ...  
}
```

Nous pouvons l'illustrer dans un programme qui va permettre de faire bouger l'image de l'étoile avec les touches du clavier. Comme les touches peuvent être enfoncées à n'importe quel moment lors de l'exécution du programme, nous devons « vérifier » cette propriété à chaque calcul de *frame* et donc dans la fonction *update()*.

Le programme ressemble alors à ceci :

```
window.onload = function() {  
  
    var game = new Phaser.Game(800, 600, Phaser.AUTO, '',  
        {preload: preload, create: create, update: update});  
    var star;  
    function preload() {  
        game.load.image('star', 'star.png');  
    }  
  
    function create() {  
        star = game.add.sprite(Math.floor(Math.random()*10)*50,  
            Math.floor(Math.random()*10)*50, 'star');  
        game.physics.arcade.enable(star);  
    }  
  
    function update() {  
        cursors = game.input.keyboard.createCursorKeys();  
        if (cursors.left.isDown) {  
            star.body.velocity.x = -100;  
        }  
        else if (cursors.right.isDown) {  
            star.body.velocity.x = 100;  
        }  
    }  
}
```

Exercice: A partir du code ci-dessus, vous allez faire bouger l'étoile dans tous les sens (haut, bas, droite et gauche) et stopper le mouvement de l'image lorsqu'aucune touche n'est enfoncée. Ceci sera la base de n'importe quel jeu qui nécessite le déplacement d'un vaisseau, d'un personnage, d'une voiture, ... La solution se trouve dans le fichier *MoveStarKeyboard.html*.

---

Nous aurions pu ne pas utiliser la « physique » proposée par *Phaser* et calculer dans la fonction *update()* la nouvelle position d'une étoile à chaque *frame*. Le code de déplacement devient alors :

```
function update () {
    if (cursors.left.isDown)
    {
        star.x = star.x - 4;
    }
    else if (cursors.right.isDown)
    {
        star.x = star.x + 4;
    }
    ...
}
```

Cette portion de code est tout à fait valide et fonctionnelle, cependant si le moteur physique n'est pas utilisé sur un élément qui rentre en collision avec le décors, par exemple, cette collision risque de ne pas être détectée par *Phaser* et donc poser des problèmes. Il est donc toujours préférable d'utiliser le changement de *velocity* plutôt que le « re-positionnement » à chaque *frame*.

## 2.9.2. Condition switch/case

Si vous avez plusieurs *if()* qui se suivent pour vérifier une série de valeurs, il est souvent plus clair et plus rapide d'utiliser la combinaison d'instructions *switch/case*. La syntaxe est la suivante :

```
switch(expression) {
    case n:
        // code à exécuter si expression est égale à n
        break;
    case m:
        // code à exécuter si expression est égale à m
        break;
    default:
        // code à exécuter par défaut des autres valeurs
}
```

## 2.10.Fonctions

En *JavaScript* une fonction se définit avec la syntaxe suivante :

```
function name(parameter1, parameter2, parameter3) {  
    // le code à exécuter lors d'un appel à la fonction  
    return value;  
}
```

### Exercices :

- Générez la position des étoiles avec des nombres aléatoires obtenus à partir d'une une fonction *randomPosition()*. Solution dans le fichier *MoveStarFunction.html*.

La fonction créée pour l'exercice ci-dessus est purement pédagogique car dans la pratique, nous utiliserons celle qui est fournie par *Phaser* : *game.rnd.between(min, max)*.

## 2.11.Tableaux

Les tableaux seront très utiles pour la réalisation des applications futures. En Javascript pour déclarer et utiliser un tableau on fait simplement :

```
var points = new Array(40);  
// crée une variable points de type tableau avec 40 éléments  
  
points[12] = 6;  
// assigne la valeur 6 au 13ème élément du tableau
```

### Exercices :

- Créez d'abord dans une boucle un tableau avec les position en X et en Y des étoiles et ensuite dans une autre boucles, placez-les à l'écran. Cet exercice est purement didactique, en production, nous ne ferions évidemment qu'une seule boucle. Solution dans le fichier *StarArray.html*.

---

Vous avez maintenant les bases nécessaires et suffisantes pour pouvoir commencer une application plus conséquente. Nous avons expliqué plus haut que nous allions réaliser un jeu complet basé sur un « vieux » shoot horizontal : *R-Type*.



Screenshot de *R-Type*<sup>6</sup>

Nous allons commencer à le programmer en procédant pas à pas :

- Affichez un vaisseau spatial à l'écran et faites le bouger avec les touches du clavier (voir solutionV1-0.html)
- Ajoutez dans la scènes des petites étoiles de manière aléatoire (voir solutionV1-1.html)
- Limitez les déplacements du vaisseau aux frontières de l'espace de jeu (voir solutionV2-2.html).

**Remarque :** vous pouvez dessiner vos *assets* vous-même, vous pouvez les décharger sur *Internet* ou vous pouvez, pour aller plus vite, utiliser ceux qui sont fournis avec ce syllabus dans le répertoire *Assets*.

---

<sup>6</sup> <http://obligement.free.fr/articles/r-type.php>

### 3. Construction de *maps*

Dans un jeu vidéo une *Map* représente la carte qui reprend les différentes parties d'un monde à explorer. Nous allons voir dans ce chapitre comment construire la carte d'un niveau de jeu en 2D. Les techniques pour les environnements 3D sont très différentes.



*Map complète du jeu Zelda: Link's Awakening<sup>7</sup>*

Pour créer un niveau complet d'un jeu en 2D, nous allons d'abord voir comment regrouper un ensemble d'éléments graphiques dans un « groupe d'objets » et nous verrons ensuite comment construire un niveau de jeu à partir de *Tiles* (tuiles graphiques).

<sup>7</sup> <https://www.videogamesblogger.com/2011/06/06/the-legend-of-zelda-links-awakening-dx-walkthrough-video-guide-3ds-virtual-console-game-boy-color-gb-classic-retro.htm>

### 3.1.Groupe d'objets

Il est souhaitable dans *Phaser* de regrouper les objets de même nature entre eux, pour par exemple, faire bouger tous les objets en même temps ou détecter une collision avec un groupe plutôt qu'avec une multitude d'objets isolés. Pour opérer ce regroupement on crée d'abord un objet (le *container*) qui contiendra l'ensemble des objets à regrouper :

```
stars = game.add.group();
```

Ensuite, il ne reste plus qu'à lui ajouter les objets, par exemple, dans une boucle :

```
for (var i = 0; i < 128; i++) {  
    stars.create(game.world.randomX, game.world.randomY, 'star');  
}
```

#### Exercice :

- Créez un groupe d'étoiles et faite-le se déplacer d'un seul bloc (voir .html).

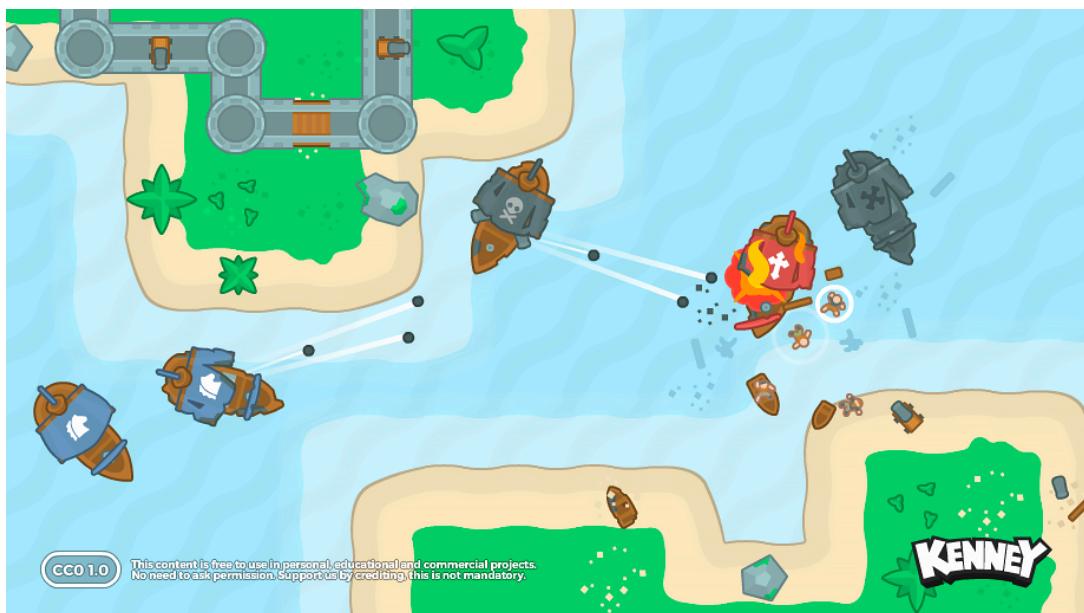
## 3.2.TileSet

Dans les jeux vidéos en 2D, on utilise fréquemment les « *TileSet* » (traduit littéralement par : un ensemble de tuiles). Il s'agit en fait d'une image qui représente un ensemble d'images plus petites (souvent carrées) qui assemblées entre elles judicieusement forment un environnement de jeu. Par exemple ce *TileSet* :



*Exemple de TileSet<sup>8</sup>*

Va permettre de réaliser un jeu tel que :



*Exemple de jeu construit à parti du TileSet*

<sup>8</sup> <http://kenney.nl/>

---

*Phaser* propose des méthodes qui permettent d'exploiter simplement ces *TileSet*. Il faut d'abord pré-charger le *TileSet* au moment du *preload()* :

```
game.load.image('tiles', 'tiles.png');
```

Le nom de l'alias (*tiles*) et du fichier (*tiles.png*) peuvent être différents de « *tiles* ».

Dans la fonction *create()*, il faut créer une « *map* » qui est une grille de *Tiles*. Cette *map* a donc un certain nombre de cases horizontalement et verticalement et chaque *Tile* à une largeur et une hauteur (en pixels). Dans notre exemple, ci-dessous, on crée une *Map* de 50 cases (horizontalement) par 40 (verticalement) avec des *Tiles* qui sont carrés (16x16 pixels) :

```
map = game.add.tilemap(null, 16, 16, 50, 40);
```

Il faut maintenant associer les *Tiles* chargés dans le *preload()*, à la *Map* que nous venons de créer avec l'instruction :

```
map.addTilesetImage('tiles', 'tiles', 16, 16);
```

Une *Map* peut contenir plusieurs couches (*layers*), nous allons donc en créer une avec l'instruction :

```
layer1 = map.create('layer1', 50, 40, 16, 16);
```

Vous remarquerez que dans notre exemple, le *layer1* recouvre complètement la *map* puisqu'il a le même nombre de cases (50x40). Il ne reste plus qu'à « remplir » la couche (*layer1*, dans notre exemple) avec les tuiles de notre *TileSet* :

```
map.putTile(2, 5, 8, layer1);
```

Dans cet exemple, nous assignons dans le *layer1*, à la case de coordonnée (5, 8), la troisième (indice 2) tuile du *TileSet*. Cette instruction se retrouvera fréquemment dans une boucle pour remplir complètement une couche d'une *map*. Dès lors, nous aurons souvent recours à la combinaison division / reste est très intéressante pour le calcul d'indices dans les tableaux ou de coordonnées dans une grille.

---

Par exemple, pour notre grille de 50 par 40 (soit 2000 cases au total) :

```
for (var i = 0; i < 2000; i++) {  
    map.putTile(Math.round(10), i%50, Math.trunc(i/10), layer1);  
}
```

### **Exercices:**

- A partir de « *solution4-0.html* », modifiez le code pour afficher le « *TileSet* » complet sur une ligne (ou deux). La solution est dans le fichier « *solution4-1.html* ».
- Générez ensuite une « *map* » aléatoire complète en utilisant le « *Tile Set* » fourni. La solution est dans le fichier « *solution4-2.html* ».

## 3.3.JSON

Si vous voulez utiliser une « *map* » fixe (qui n'est pas générée au chargement du jeu) vous pouvez la créer avec un logiciel tel que *Tiled*<sup>9</sup>, la sauvegarder dans un format *json* ou *xml* (extension *.tml*) et ensuite l'utiliser dans *Phaser* de la manière suivante :

```
function preload() {  
    game.load.tilemap('map1', 'map1.json', null,  
                      Phaser.Tilemap.TILED_JSON);  
    game.load.image('tiles', 'sci-fi-tiles.png');  
}  
  
function create() {  
    game.stage.backgroundColor = '#787878';  
    map = game.add.tilemap('map1');  
    map.addTilesetImage('SciFiTiles', 'tiles');  
    layer = map.createLayer('Map1');  
    layer.resizeWorld();  
}
```

---

<sup>9</sup> <http://www.mapeditor.org/>

```

...
    "height":40,
    "name":"Map1",
    "opacity":1,
    "type":"tilelayer",
    "visible":true,
    "width":50,
    "x":0,
    "y":0
  ],
  "nextobjectid":1,
  "orientation":"orthogonal",
  "renderorder":"right-down",
  "tileheight":16,
  "tilesets":[
    {
      "columns":28,
      "firstgid":1,
      "image":"sci-fi-tiles.png",
      "imageheight":32,
      "imagewidth":448,
      "margin":0,
      "name":"SciFiTiles",
      "spacing":0,
      "tilecount":56,
      ...
    }
  ]
}

```

### Attention :

- dans la méthode `add.tilemap()`, le paramètre doit avoir la même valeur que le *TileSet name* dans le logiciel *Tiled*
- dans la méthode `addTilesetImage()` le paramètre doit avoir la même valeur que le nom du calque dans le logiciel *Tiled*

---

Exercice « R-Type » : Créer à partir d'un *Tileset* donné (<http://www.spriter-resource.com/resources/sheets/49/52083.png>) un environnement aléatoire et cohérent pour notre « Space Shooter ».

Nous allons procéder par étape :

- choisir dans le *Tileset* une tuile qui ne nécessite pas de voisin et faites en bas de l'écran une « ligne » à partir de ce bloc (voir solutionV3-1)
  - avec la même tuile faites en bas de l'écran une « courbe » qui souligne un chemin (voir solutionV3-2)
    - faites la même chose en haut de l'écran (voir solutionV3-3)
    - vous devez maintenant gérer le fait que les deux « lignes » du dessus et dessous peuvent parfois s'entre-croiser (voir solutionV3-4)
      - essayez de démarrer vos « lignes » ailleurs que dans les coins (voir solutionV3-5)
      - adaptez la position initiale du vaisseau en fonction du chemin (voir solutionV3-6)
      - construisez vos lignes sous forme d'un « couloir » d'une taille définie au départ, cela nous permettra de mettre un niveau de difficulté en réduisant la taille du couloir, attention à la position de départ du vaisseau (voir solutionV3-7)

Nous allons améliorer l'environnement aléatoire de notre « Space Shooter », en le rendant plus esthétique et encore plus cohérent.

Nous allons procéder par étape :

- essayez d'abord de trouver dans le « Tile Set » des tuiles qui peuvent s'enchaîner côté à côté et construisez une suite de tuiles cohérentes et aléatoires :
  - \* première étape : on va d'abord créer un tableau de « Tiles » qui contiendra la base de notre chemin (voir solutionV3-8)
  - \* deuxième étape : on va maintenant essayer d'adoucir les angles en remplaçant les trous entre les blocs (voir solutionV3-9)
  - \* troisième étape : on va finalement remplir les espaces vides en haut et en bas de l'écran (voir solutionV3-10)
- positionnez un ennemi au sol de manière aléatoire mais conservez sa position en mémoire (voir solutionV3-11)

## 4. Evénements temporels

Il est souvent intéressant de générer des événements temporels, pour par exemple, faire bouger automatiquement un ennemi ou lui faire tirer des missiles de manières régulières. En *Javascript* la syntaxe est la suivante :

```
setTimeout(myFunction, timeInMilliseconds)

function myFunction() {
    // code to execute every timeInMilliseconds
}
```

Ajouter ici des exercices en fonction des besoins.

Il existe également une fonction plus polyvalente dans l'environnement *Phaser* :

```
game.time.events.repeat(Phaser.Timer.SECOND * 2, 20, myFunction,
this);

function myFunction() {

    // code to execute 20 times every 2 SECONDS

}
```

**Exercice :** reprenez l'exercice qui remplit l'écran d'étoiles et générez ces étoiles une à une à une cadence de une étoile toutes les deux secondes (solution dans exercice2-4.html).

Si vous devez passer des paramètres à la fonction, vous devez les spécifier après le *this* :

```
game.time.events.repeat(Phaser.Timer.SECOND * 2, 10, createStar,
this, 50, 10);

...
function createStar(xdist, ydist) {
...
}
```

---

**Exercice « R-Type » :** Nous avons créé une base ennemie, nous allons la faire tirer des missiles en passant par plusieurs étapes :

- pour commencer créez un tir depuis n'importe quel endroit sur l'écran (position aléatoire) et dans n'importe quelle direction (voir solutionV4-1.html). Pour ce faire inspirez-vous du code suivant qui permet d'associer une « bullet » au moteur physique de Phaser, ce qui permet ensuite de lui appliquer une vitesse (*velocity*).

```
game.physics.arcade.enable(bullet);  
bullet.body.velocity.x = 100;
```

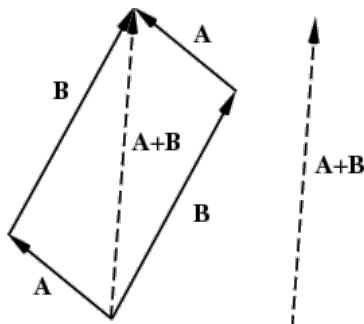
- recommencez l'opération toutes les cinq secondes mais cette fois depuis la station ennemie (voir solutionV4-2.html)
  - vous allez maintenant tirer en direction du vaisseau du joueur (voir solutionV4-3.html). Il est utile pour cet exercice de se souvenir de la théorie de base des vecteurs.

# 5. Physique

Pour aborder les notions liées à la physique dans *Phaser* (notions de force et de collision), nous allons commencer par faire un rappel sur la théorie des vecteurs qui est indispensables pour la bonne compréhension du chapitre.

## 5.1. Vecteurs

### 5.1.1. Addition et soustraction



*Addition de deux vecteur*

Pour additionner deux vecteurs  $A$  et  $B$  on utilise la règle dite du « parallélogramme ». Il suffit de placer l'origine d'un vecteur sur l'extrémité de l'autre vecteur. La somme est le vecteur qui part de l'origine de ce dernier et se termine à l'extrémité du premier. En coordonnées cartésiennes, il suffit d'additionner les coordonnées des deux vecteurs :

$$\begin{aligned} \text{Si } A &= (X_a, Y_a) \\ \text{et } B &= (X_b, Y_b) \\ \text{alors } A+B &= (X_a+X_b, Y_a+Y_b) \end{aligned}$$

Pour une soustraction il suffit d'ajouter le vecteur opposé:

$$A-B = A+(-B)$$

ce qui revient en coordonnées scalaires à faire

$$A-B = (X_a-X_b, Y_a-Y_b)$$

---

### **Exercices :**

- Vous remarquerez que plus votre vaisseau est proche de la base ennemie et plus la vitesse du missile est petite. Il faudrait donc normaliser la vitesse du tir, c'est à dire conserver toujours la même vitesse de tir quelle que soit la distance entre le vaisseau et l'ennemi (voir solutionV4-4.html). pour ce faire il serait utile de faire un petit rappel sur la norme des vecteurs et le calcul de la distance entre deux points.

### **5.1.2.Norme d'un vecteur**

La norme d'un vecteur est une proportion de sa longueur. Elle peut se calculer à l'aide de ses coordonnées dans un repère orthonormé grâce au théorème de *Pythagore*. En coordonnées scalaires la norme du vecteur A, notée :

$$\|A\| = \sqrt{(X_a^2 + Y_a^2)}$$

### **5.1.3. Distance entre deux points**

En combinant les deux points précédents, on peut donc calculer la distance entre deux points en faisant :

$$\sqrt{((X_a - X_b)^2 + (Y_a - Y_b)^2)}$$

---

## 5.2.Forces

## 5.3.Collisions

Pour détecter les collisions avec les différents *layers*, il va falloir stipuler au moteur de jeu quels sont les *tiles* qui provoquent des collisions avec la méthode `SetCollisionBetween(,)`.

```
map = game.add.tilemap(null, 16, 16, 50, 40);
map.addTilesetImage('tiles', 'tiles', 16, 16);

map.setCollisionBetween(1, 2055);
```

Ensuite, comme nous l'avons déjà fait précédemment, il suffit d'ajouter un *collider* entre le décor et le vaisseau :

```
game.physics.arcade.collide(player, layer1,
collisionLayerPlayer, null, this);
```

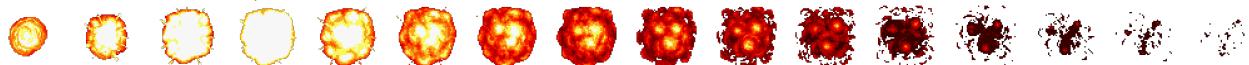
### Exercice :

- Détectez maintenant les collisions entre les missiles et le vaisseau pour avoir enfin un programme avec un enjeu (voir solutions V4-5.html et V4-8.html).

# 6. Animations

## 6.1. Sprite animé

Pour animer un *Sprite*, il faut créer ou télécharger une *SpriteSheet* qui contient l'animation (un ensemble d'images qui affichées les unes derrière les autres donnent une impression d'animation).



*Exemple de SpriteSheet*

Au niveau du code, il faut dans la fonction *preload* charger l'image en précisant que c'est un *SpriteSheet* avec la taille de chacun des éléments de l'image (dans l'exemple suivant 128 x 128) et le nombre d'image (dans l'exemple suivant 16) :

```
function preload() {  
    game.load.spritesheet('boom', 'explosion.png', 128, 128, 16);  
}
```

Ensuite il faut créer une animation basée sur ce *SpriteSheet* et la faire jouer avec un *play* (la valeur 50 de l'exemple correspond à la vitesse de lecture de l'animation) :

```
function create() {  
    explosion = game.add.sprite(40, 100, 'boom');  
    explosion.animations.add('explode');  
    explosion.animations.play('explode', 50, true);  
}
```

### Exercice :

- générez une animation qui montre une explosion du vaisseau (voir solutionV4-7.html).

# 7. Classes et objets

Nous avons déjà vu que le *JavaScript* permet l'utilisation d'objets, nous allons voir maintenant comment définir nos propres classes avec leurs propriétés et leurs méthodes et créer à partir de celles-ci nos propres objets.

## 7.1.Définition d'une classe

Pour définir une classe, nous allons d'abord définir quelques propriétés ainsi qu'un constructeur, une méthode particulière, qui nous permettra de créer des instances de la classe : les objets.

```
function Ennemy(life, experience) {  
    this.life = life;  
    this.experience = experience;  
}
```

Pour créer un objet à partir de cette classe, nous allons utiliser l'instruction « *new* ».

```
var orcEnnemy = new Ennemy(5, 17);
```

On peut définir un objet encore plus rapidement (sans créer de classe préalablement) en utilisant la syntaxe suivante :

```
var orcEnnemy = {life:"5", experience: »17»};
```

## 7.2.Propriétés

Nous pourrons ensuite accéder aux propriétés des objets en y faisant référence comme nous l'avons déjà vu plus haut.

```
alert(orcEnnemy.life);
```

Il est parfois commode de pouvoir accéder aux propriétés de cette manière là :

```
alert(orcEnnemy["life"]);
```

## 7.3.Méthodes

Il y a deux manières de définir une méthode dans une classe : dans la méthode « constructeur » ou en utilisant les « prototype ». En général quand il s'agit d'une classe personnelle, on définit les méthodes directement dans le constructeur.

```
function Ennemy(life, experience) {  
    this.life = life;  
    this.experience = experience;  
  
    this.hitted = function() {  
        this.life -- ;  
    };  
}
```

Par contre lorsqu'on veut « augmenter » une classe existante avec de nouvelles fonctionnalités (méthodes) on utilise la définition via prototype.

```
Person.prototype.hitted = function() {  
    this.life -- ;  
}
```

Exercice « R-Type » : Nous allons maintenant réécrire le code plus « proprement » et dans une optique « *orienté objet* » :

- on va transférer les éléments graphiques dans un répertoire « assets » et y faire référence au niveau du « path » de manière relative (voir solutionV5-1.html)
- notre vaisseau est déjà un objet on peut simplement lui ajouter une nouvelle propriété qui va stocker le nombre de collisions avec un missile (voir solutionV5-2.html)
- on va ensuite créer des ennemis mobiles et leur donner des trajectoires (voir solutionV5-3.html)

Idées pour ajout :

- faire clignoter le player quand il est touché par un missile (avec alpha ? et pas linéaire ?)
- ajouter un score (nombre de vaisseaux détruits, par exemple)

## 7.4.Groupe d'objets

Dans l'exercice précédent seule le dernier missile envoyé percute le vaisseau. C'est parce que vous redéfinissez la variable à chaque création d'occurrence d'un nouveau missile. Pour corriger ce problème, nous pouvons créer un « pool » de missile avec les *group* dans *Phaser*.

```
bullets = game.add.group();
bullets.enableBody = true;
bullets.physicsBodyType = Phaser.Physics.ARCADE;
bullets.createMultiple(30, 'etoile');
bullets.setAll('outOfBoundsKill', true);
bullets.setAll('checkWorldBounds', true);
```

### Exercice :

Créez un « groupe » de missile de manière à pouvoir détecter les collisions avec chaque missile et pas seulement le dernier tiré (voir solutionV4-6.html).

## 7.5.Gestion des groupes d'objets

Pour éliminer les « *bullets* » qui sont sorties de l'écran quand on *scroll* :

```
bullets.forEachAlive(function(bullet) {
    if(bullet.x > xLimit) bullet.kill();
}, this);
```

# 8. Camera

## 8.1. Scrolling

Pour faire défiler une séquence de jeu, il faut créer la caméra principale qui va « parcourir » l'espace du jeu.

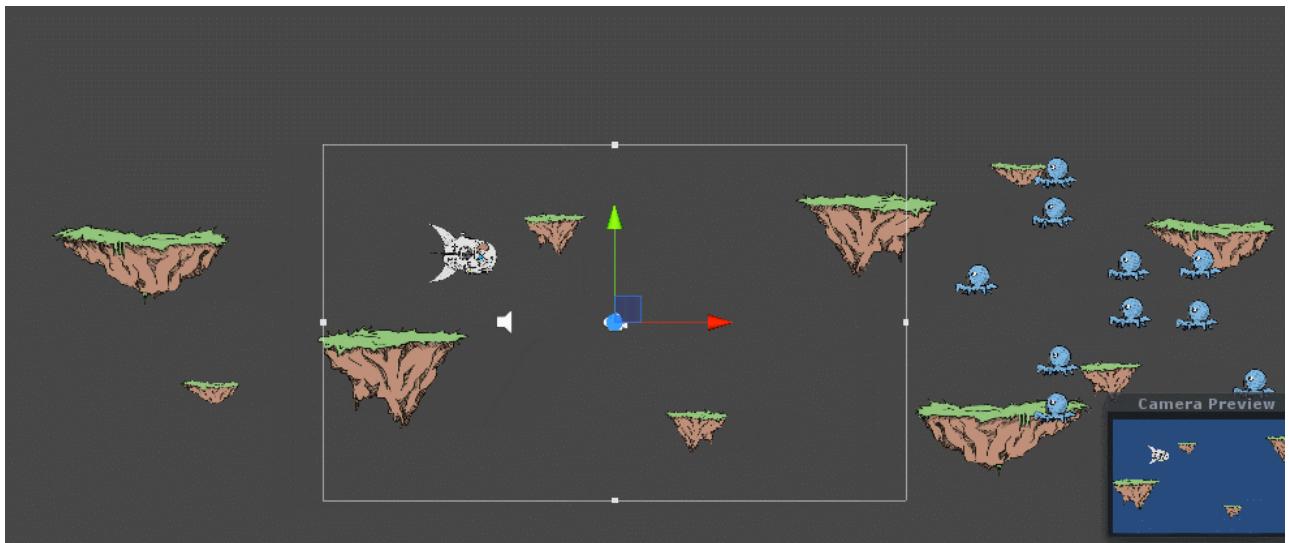


Illustration du scrolling par mouvement de caméra<sup>10</sup>

Cette création se fait comme nous l'avons déjà fait avec :

```
var game = new Phaser.Game(800, 320, ...)
```

C'est l'espace de jeu (la *map*) qui doit être plus grand que ce qui est « vu » par la caméra :

```
layer1 = map.create('layer1', 200, 20, 16, 16);
```

Dans cet exemple on a un espace horizontal de  $200 \times 16 = 3200$  pixels alors que la caméra ne montre que 800 pixels.

Ensuite, pour faire défiler l'écran horizontalement on va, par exemple, à chaque `update()` incrémenter la position de la caméra :

```
game.camera.x += 1;
```

<sup>10</sup> <http://pixelnest.io/tutorials/2d-game-unity/parallax-scrolling/>

## 9. Audio

Pour insérer du son dans un jeu il suffit comme pour l'insertion d'images de d'abord (dans la fonction `preload`) pré-charger le son :

```
game.load.audio('explosionSound', './explosion.wav');
```

Ensuite dans la partie de création du jeu (fonction `create`) on doit créer un objet qui va contenir ce son et permettre sa gestion (jouer le son, boucler le son, ...):

```
explosionSound = game.add.audio('explosionSound');
```

Pour enfin pouvoir être exploité dans le jeu lui même :

```
explosionSound.play();
```

# 10. Sauvegarde de données

Si vous désirez sauvegarder des informations sur le disque en local qui seront réutilisées à chaque démarrage de l'application (le meilleur score réalisé, par exemple), vous pouvez utiliser les instructions suivantes :

```
localStorage.setItem("BestScore", "" + bestScore);
```

pour enregistrer la valeur d'une variable, et

```
bestScore = parseInt(localStorage.getItem("BestScore"));
```

pour relire cette valeur, au démarrage, par exemple.

# 11. Intégration dans une page HTML

Nous avons vu plus haut que l'application JavaScript est intégrée dans une page HTML dans le genre :

```
<html>
  <head>
    <title>Exercise XXX</title>
    <script src=<> ../phaser-x.y.z/build/phaser.min.js"></script>
  </head>
  <body>
    <script type="text/javascript">

      // Votre code ici

    </script>
  </body>
</html>
```

Pour centrer notre application Phaser au milieu de la page horizontalement il ne faut pas le faire avec du CSS mais simplement en ajoutant dans la méthode `create` la ligne suivante :

```
game.scale.pageAlignHorizontally = true;
```

On peut ensuite « décorer » notre environnement de jeu avec un background de couleur, avec une image, avec du texte supplémentaire, ...

```
<body background="/PhaserTutorial/R-Type/Assets/space.png">
<div style="text-align:center; font-size: x-large; font-family: sans-serif; color: #FFFFFF; margin-top: 20; margin-bottom: 20">
R-Type @ Rudi Giot - 2016</div>
```

# 12.Organisation du code

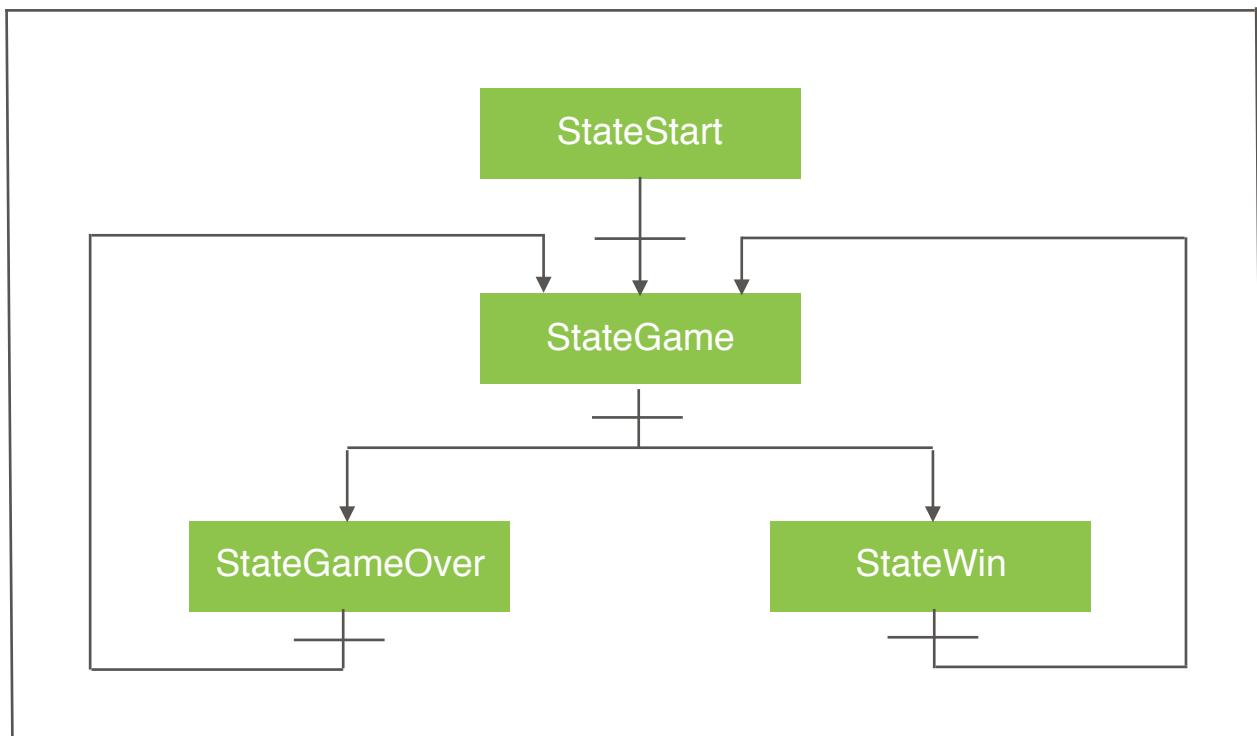
## 12.1.Machine à états finis

Dans la programmation de jeux vidéos on utilise très fréquemment des « *finite-state machine* » aussi appelé « *automate fini* ». Une machine (un programme dans notre cas) ne sera jamais que dans un et un seul état à un moment donné. Cet état est appelé l'état courant (current state). La machine passe d'un état à un autre lors d'une transition suite à un événement ou une condition réalisée.

Dans notre jeu, nous allons créer des états pour chaque « écran » du jeu. Pour :

- le démarrage (écran d'accueil),
- le jeu à proprement parlé
- le « Game Over »
- le « You Win »

RTypeGame



---

Au niveau du code, nous allons devoir d'abord créer un objet qui contiendra nos différents états : *RTypeGame*.

```
var RTypeGame = {};
```

Nous allons ensuite décrire chacun des états en deux parties. D'abord la déclaration des variables globales de l'état :

```
RTypeGame.StateStart = function (game) {
    this.playText;
};
```

Et ensuite la déclaration des différentes méthodes utilisées dans cet état :

```
RTypeGame.StateStart.prototype = {

    preload: function () {

        game.load.image('space', 'karamoon.png');
        game.load.image('player', 'ship.png');
        game.load.bitmapFont('command', 'command.png', 'command.xml');

    },

    mouseOver: function () {
        //alert("Stop");
        this.playText.fill = "#ff0000";
    },
    ...
}
```

Cette opération (les deux parties) doit être réalisée pour chacun des états. Pour terminer nous allons créer le jeu, ajouter les différents états au jeu et définir l'état d'entrée (le premier état de notre automate) :

```
var game = new Phaser.Game(800, 320, Phaser.CANVAS, 'R-Type');

game.state.add('StateGame', RTypeGame.StateGame);
game.state.add('StateStart', RTypeGame.StateStart);

game.state.start('StateStart');
```

**Exercice :** Réaliser différents écrans pour le jeu R-Type (Solution 6-2.html).

## 12.2. Segmentation du code en plusieurs fichiers

Pour des raisons d'organisation et de lisibilité du code, il est pratique de scinder son programme en plusieurs fichiers indépendants. Il faudra ensuite y faire appel (une sorte d'*include*) dans un fichier « fédérateur » qui ressemble à ceci :

```
<html>
  <head>
    <meta charset="UTF-8" />
    <title>R-Type Final Version</title>
  </head>
  <body background=<< /PhaserTutorial/R-Type/Assets/space.png">></body>

    <script src="/phaser-x.y.z/build/phaser.min.js"></script>

    <script src="Global.js"></script>
    <script src="GameMainState.js"></script>
    <script src="GameOverState.js"></script>
    <script src="GameWinState.js"></script>
    <script src="GameStartState.js"></script>

    <script type="text/javascript">

      var game = new Phaser.Game(800, 320, Phaser.CANVAS, 'R-Type');

      game.state.add('StateGame', RTypeGame.StateGame);
      game.state.add('StateStart', RTypeGame.StateStart);
      game.state.add('StateGameOver', RTypeGame.StateGameOver);
      game.state.add('StateGameWin', RTypeGame.StateGameWin);

      game.state.start('StateStart');

    </script>
  </body>
</html>
```

---

### **Projet libre et individuel**

Ce projet s'étalera tout au long de la formation. Vous devez d'abord imaginer un jeu en 2D simple qui servirait de promotion à une marque ou une entreprise. Il pourrait s'agir d'une société pour la faire connaitre, ou d'un produit pour en faire la promotion, ... La société ne doit pas être nécessairement commerciale, ça pourrait être une association humanitaire, une ASBL, ... Vous pouvez choisir un produit, une marque, une cause qui vous correspond. Quand le choix est fait, vous imaginez une application *ludique* qui se joue en 3-4 minutes maximum et qui au final promeut votre « marque ». Ensuite, vous :

- Faites l'analyse sommaire du jeu et donnez les spécifications de base de votre application.
- Cherchez ou créez vos « *Assets* » de base à partir de quoi vous allez réaliser vos premières maquettes.
- Définissez ensuite les différentes étapes de développement (versions) de votre jeu et essayez de les planifier sur le nombre de jour dédiés à la formation.