

Phaser

Jeux vidéo 2D en Javascript

Rudi Giot - 27 janvier 2021



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

1. Introduction	7
1.1.HTML5 et Javascript	7
1.2.Phaser	7
1.3.Serveur HTTP	8
1.4.Sandbox	10
1.5.Installation de Phaser	10
1.6.Exercice « fil rouge »	11
2. Les bases	12
2.1.Structure d'un programme	12
2.2.Machine à états	15
2.3.Variables	17
2.4.Commentaires	19
2.5.Débogage	19
2.6.Boucles	20
2.7.Physique	22
2.8.Saisie clavier	23
2.9.Instructions conditionnelles	23
2.10.Fonctions	26
3. Construction de maps	29
3.1.Groupe d'objets	30
3.2.TileSet	32
3.3.Tiled	36
3.4.Tableaux	40
4. Evénements temporels	42
4.1.Evénements déclenchés après un délai	42
4.2.Evénements générés en rafale	42
5. Physique	44
5.1.Vitesse	44
5.2.Collisions	47

5.3.Gravité	49
6. Animations	50
6.1.Avec timer	50
6.2.Avec tween	50
6.3.Sprite animé	51
6.4.Animation bouclée	52
7. Classes et objets	53
7.1.Définition d'une classe	53
7.2.Instanciation d'un objet	53
7.3.Propriétés	53
7.4.Méthodes	54
8. Camera	55
9. Audio	56
10.Interface utilisateur	58
10.1.Ecran d'accueil	58
10.2.Interactivité	58
10.3.Informations « in game »	59
11.Sauvegarde de données	60
12.Intégration dans une page HTML	61
13.Organisation du code	62
13.1.Machine à états finis	62
13.2.Segmentations du code en plusieurs fichiers	65
14.Publier sur itch.io	67
15.Atelier de Noël	70
15.1.Introduction	70
15.2.Images	70
15.3.Boucle et nombres aléatoires	71
15.4.Fonction update() et transparence	71
15.5.Tween	72

15.6.Pool d'objets	72
15.7.Jouer une musique	73
15.8.Boutons interactifs	74
15.9.Textes	74
15.10.Personnalisation	74
15.11.Publication sur Internet	74
16.Atelier Quiz	75
16.1.Introduction	75
16.2.Images	75
16.3.Textes	76
16.4.Boutons interactifs	76
16.5.Physique	77
16.6.Objets JSON	78
16.7.Tableaux d'objets	78
16.8.Tween	79
16.9.Son	79
16.10.Timer	80
16.11.Font personnelle	80
17.Atelier Frogger	81
17.1.Introduction	81
17.2.Images	81
17.3.Interactivité clavier	82
17.4.Détection de collisions	83
17.5.Moteur physique	83
17.6.Son	84
17.7.Bouton interactif	84
18.Atelier R-Type	85
18.1.Introduction	85
18.2.Images	85

18.3.Interaction clavier	86
18.4.Collisions	86
18.5.Timer	88
18.6.Images animées	88
18.7.Son	89
18.8.Tileset	89
18.9.Orientation d'un tir	90

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 *assets* (éléments graphiques et sonores) 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 d'autres contextes, côté serveur, comme dans *Node.js*, par exemple. Il existe de nombreux « *framework* » et librairies qui facilitent la programmation en *JavaScript*, *JQuery*, *Three.js* et *Phaser* par exemple.



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 « déboguées ». De plus, le site <http://phaser.io> comporte de nombreux exemples, tutoriels et un forum très actifs. *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.

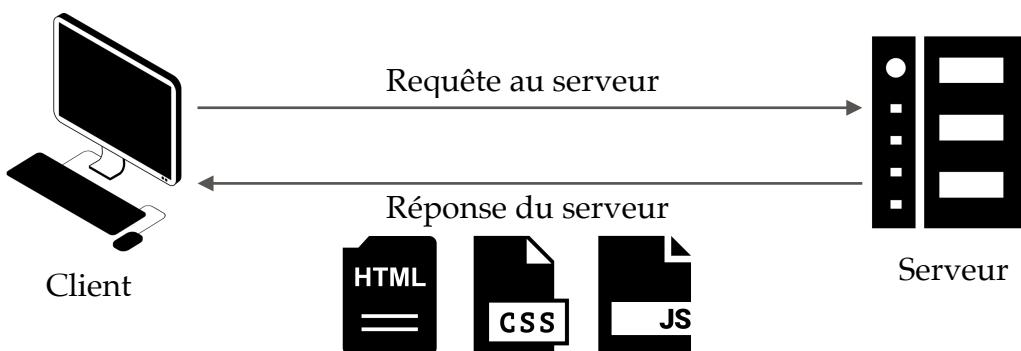


Schéma d'une communication utilisant l'HTTP

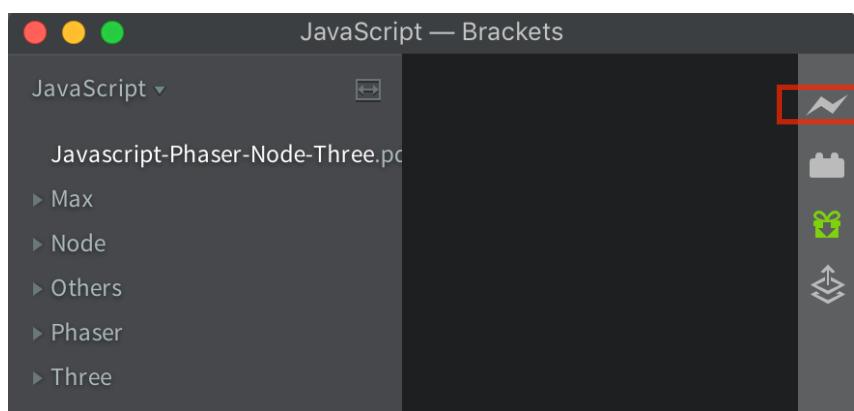
L'installation d'un serveur Web « opérationnel » (exposé sur *Internet*) est une opération très complexe. En effet, sa sécurisation est primordiale et nécessite des connaissances approfondies dans le domaine des réseaux. Heureusement, dans votre cas, vous n'avez besoin que d'un serveur *HTTP* de développement, un service qui s'exécute localement sur votre 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). Vous pouvez choisir la méthode qui vous paraît être la plus pratique.

1.3.1. _AMP

Nous pouvons utiliser une distribution libre et gratuite du serveur *Apache* qui est intégré dans *WAMP*¹ (pour *Windows*), *MAMP*² (pour *MacOS*) ou *LAMP*³ (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

¹ <http://www.wampserver.com>

² <https://www.mamp.info>

³ <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 rapidement ses limites. Elle est idéale pour tester rapidement des portions de code ou développer un petit prototype. 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 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.

L'installation « complète » de *Phaser* n'est pas nécessaire pour développer un projet. Nous pourrions en effet, juste nous contenter de télécharger le fichier *phaser.min.js* qui contient toute la librairie nécessaire, les autres fichiers du package contiennent les exemples et la documentation. Nous verrons même plus loin qu'on pourrait encore plus simplement juste faire référence à la librairie hébergée sur un autre site, sans devoir la télécharger.

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* original⁴

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

2. Les bases

2.1. Structure d'un programme

Une application *Phaser* s'intègre 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/dist/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 toujours plus ou moins de la manière suivante :

```
let config = {
  type: Phaser.AUTO,
  width: 800,
  height: 600,
  physics: { default: 'arcade' },
  scene: { create: create }
};

var game = new Phaser.Game(config);

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*

Pour être ordonné, nous allons dissocier le code *HTML* du code *JavaScript*. Ce sera plus clair et vos projets seront plus lisibles. Nous allons donc créer avant de nous lancer dans des programmes plus complexes dans la réalisation d'un *template* que nous réutiliserons dans chaque projet. Nous allons d'abord créer un répertoire pour ce modèle, nous l'appelons *template*, il va contenir un fichier et deux répertoires :

- assets (répertoire)
- index.html (fichier)
- js (répertoire)

Le répertoire *js* contiendra nos codes *JavaScript*, les deux fichiers principaux seront :

- phaser.min.js
- game.js

Le répertoire *assets* contiendra lui même plusieurs sous-répertoires :

- images
- audio
- animations
- ...

Exercice :

- Créez les répertoires et fichiers dont nous venons de parler, la « fondation » de chacun de nos futurs projets, et sauvegarder l'ensemble dans un répertoire « template » qui servira de base pour vos futures applications.

A partir de maintenant, à chaque fois que vous créez un nouveau projet, il suffira de dupliquer le répertoire modèle que nous venons de réaliser, de le renommer et ensuite de remplir le fichier *game.js* avec votre code.

Vous pourriez penser qu'il serait plus intelligent de mettre le fichier *phaser.min.js* dans un autre répertoire et y faire référence à partir de tous les exercices que nous allons faire plutôt que de le dupliquer à chaque fois dans notre projet, vous auriez sans doute un peu raison. Sauf que, lorsque vous créez un projet dans une version de *Phaser*, vous voulez toujours faire référence à la librairie dans cette version là. Certaines méthodes changent parfois d'une version à l'autre et il est toujours prudent de conserver la version de la librairie qui correspond à celle qui a été utilisée pour créer votre code.

Le fichier *HTML* de base (*index.html*) ressemble alors à ceci :

```
<!doctype html>
<html>
  <head>
    <script src=".js/phaser.min.js"></script>
  </head>
  <body>
    <script src=".js/game.js"></script>
  </body>
</html>
```

Et le fichier *game.js* contient un code de base du genre :

```
let config = {
  type: Phaser.AUTO,
  width: 800,
  height: 600,
  scene: { create: create }
};

var game = new Phaser.Game(config);

function create ()
{
  // Le code pour la création de la scène
}
```

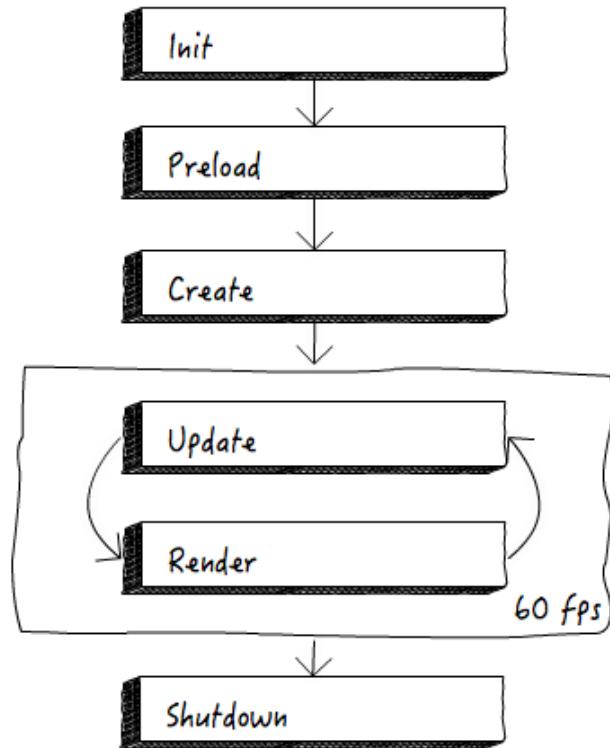
2.2.Machine à états

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* ait 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 changez d'état dans votre jeu. Nous verrons son utilité tout à la fin de ce document.

En résumé, graphiquement, on peut représenter la machine à états de *Phaser* de la manière suivante :



Machine à état de *Phaser*⁵

Les fonctions utilisées doivent être déclarées dans l'objet « *config* » (on utilise généralement ce nom) de votre application :

```
let config = {  
    ...  
    scene: {  
        preload : preload,  
        create: create,  
        update : update  
    }  
};
```

Exercice : Intégrez ces nouveaux éléments (y compris la fonction associée dans le bon fichier de votre répertoire *template*.

⁵ <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. Par exemple, les variables `_life`, `$player` ou `gun2` sont valides alors que `12bullets`, `+mana` ou `&cute` ne le sont pas.

2.3.1. Type

Le « **typage** » en *Javascript* est dit « **faible** ». Cela signifie que l'on peut déclarer une variable simplement en utilisant, en fonction de la version de *Javascript*, les instructions « `var` », « `const` » ou « `let` », sans spécifier si c'est un entier, une chaîne de caractères ou un booléen. On utilisera dans la suite de ce cours le mot-clé « `var` » en sachant qu'il faudra lentement migrer vers une utilisation des « `const` » et « `let` » dans le cas particulier des variables locales et des constantes.

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 « + ».
- On peut inclure une chaîne de caractère à l'intérieur d'une autre chaîne de caractère de la manière suivante (attention à l'utilisation des caractères spéciaux *backtick*, dollar et accolades) :

```
var name = "Rudi";
alert (`Hello, ${name}, how are you ?`);
```

2.3.2. *let* et *var*

Actuellement, *JavaScript* propose deux syntaxes pour déclarer une variable : *let* et *var*. On utilise le *var* pour déclarer les variables globales alors que *let* va limiter l'utilisation de la variable au contexte dans lequel elle est définie (variable locale).

2.3.3. Constante

Vous avez également en *JavaScript* la possibilité de déclarer des variables dont la valeur ne change pas : des constantes. La syntaxe est très simple :

```
const maxHealth = 5;
```

2.3.4. Casting

Attention aux changements de type de valeur. Par exemple, quand une variable contient un entier et que vous voulez l'afficher il faut d'abord la transformer en chaîne de caractère.

```
int age = 25;
alert ("You are " + age.toString() + " years old");
```

Cependant la plupart du temps *JavaScript* réalise cela pour vous de manière implicite:

```
int age = 25;
alert (`You are ${age} years old`);
```

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. Débogage

Pour déboguer un code *JavaScript*, il est faut utiliser les « options développeurs » de votre Browser, dans *Chrome* et *Firefox*, elles sont nombreuses et très pratiques (*console*, *elements*, *timeline*, ...). Il est également toujours préconisé de travailler en « mode privé » de manière à éviter que le « cache » n'empêche le rafraîchissement complet des images ou du son 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 les instructions *alert()* pour réaliser un *breakpoint* ou *console.log()* pour afficher la valeur d'une variable à un moment donné du programme dans la console du browser :

```
alert("Variable i : " + i);  
console.log("Variable i : " + i);
```

Nous verrons aussi plus loin comment utiliser la fonction *render()* pour afficher l'état des 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*. Une solution est proposée dans le fichier *VariablesAlert.html*.

2.6.Boucles

2.6.1. Boucle « for »

La boucle « for » a la syntaxe suivante :

```
for (initialisation; condition; incrémentation){  
    ...  
}
```

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

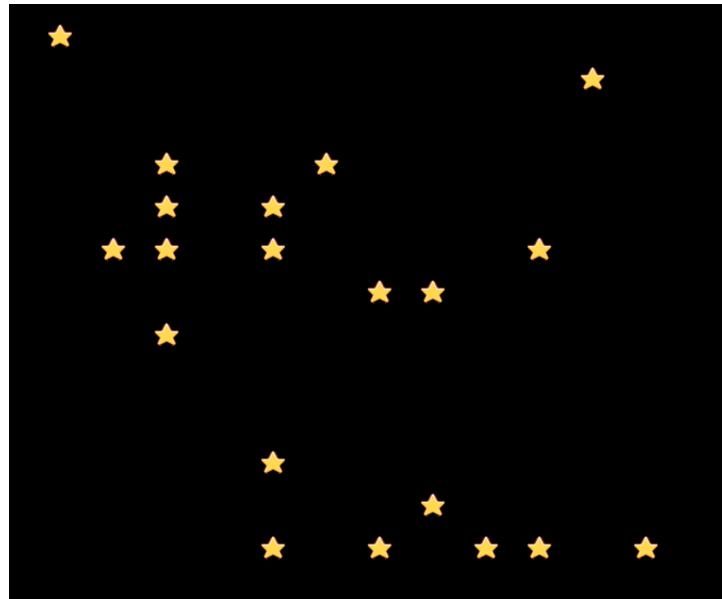
```
var config = {  
    type: Phaser.AUTO,  
    width: 800, height: 600,  
    physics: { default: 'arcade' },  
    scene: {  
        preload : preload,  
        create: create  
    }  
  
var game = new Phaser.Game(config);  
  
function preload() {  
    this.load.image('etoile', 'star.png');  
}  
  
function create() {  
    for (var i=0; i<10; i++){  
        this.add.sprite(20 + i * 50, 20 + i * 50, 'etoile');  
    }  
}
```

Remarquez dans le code ci-dessus comment afficher une image. Il faut d'abord pré-charger l'image avec la méthode `this.load.image()` qui l'associe à une référence ('etoile' dans l'exemple) et ensuite l'afficher avec la méthode `this.add.sprite()` en spécifiant sa position à l'écran (coordonnées x, y) et le nom de la référence (aussi appelé « label ») à l'image.

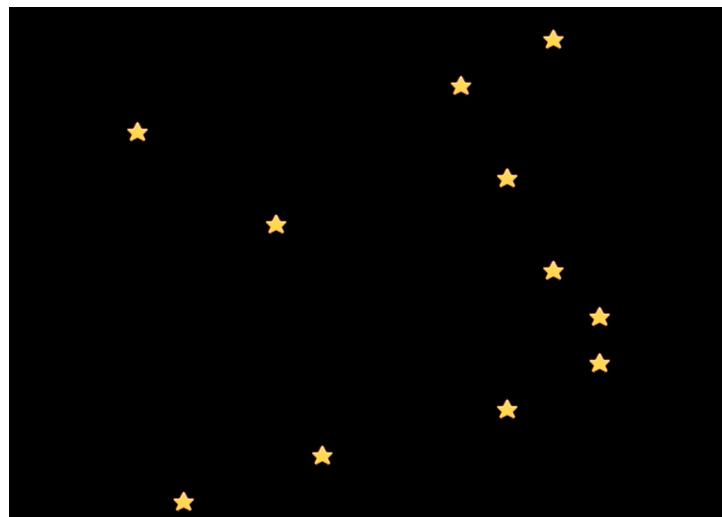
Dans l'exemple donné l'image (le fichier `png` doit se trouver dans le même répertoire que vos trois fichiers précédents. Dans la pratique, nous avons vu plus haut que l'on préfère avoir un répertoire nommé `assets` qui contiendra tous les éléments du jeu.

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 répertoire *RandomStar*.



- 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 répertoire *RandomStarInLine*.



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 répertoire *RandomStarWhile*.

2.7.Physique

L'intérêt d'utiliser d'une librairie comme *Phaser* réside dans le fait qu'elle 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, entre autres, de gérer la physique des objets pour vous. Nous allons, par exemple, afficher une étoile et la transformer en une étoile filante en lui donnant de la 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 les axes des X et des Y) de la valeur désirée :

```
function preload() {  
    this.load.image('star', 'star.png');  
}  
  
function create() {  
    var star = this.physics.add.image(400, 100, 'star');  
    star.setVelocity(100, 0);  
}
```

Exercice :

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

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 = this.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 :

```
let config = {
    type: Phaser.AUTO,
    width: 800, height: 600,
    physics: { default: 'arcade' },
    scene: { preload: preload, create: create, update : update }
};

let game = new Phaser.Game(config);
let star, cursors;

function preload() {
    this.load.image('star', '../Assets/star.png');
}

function create() {
    cursors = this.input.keyboard.createCursorKeys();
    star = this.physics.add.image(50, 50, 'star');
}

function update() {
    if (cursors.left.isDown) {
        star.setVelocityX(-100);
    } else if (cursors.right.isDown) {
        star.setVelocityX(100);
    }
}
```

Remarque : On peut également changer la vitesse en utilisant la propriété :

```
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 répertoire *MoveStarKeyboard*.

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 (fichier *MoveStarKeyboardAlt.html*) , cependant, dans ce cas, comme vous n'utilisez pas le moteur physique (le changement de vitesse) sur un élément et qu'il rentre en collision avec les décors, par exemple, elle ne sera pas détectée par *Phaser*. 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 répertoire *MoveStarFunction*.
- Essayez de modifier les propriétés *scaleX (float)*, *scaleY (float)*, *displayWidth (int)*, *flipX (boolean)*, *flipY (boolean)* et la méthode *setScale(float, float)* d'un *sprite* et regardez l'effet provoqué. ces méthodes seront peut-être utiles dans la suite de vos projets.
- Essayez les méthodes *setAngle(float)*, *setRotation(float)* et *setOrigin(float, float)* d'un *sprite* et observez les effets combinés de ces trois méthodes. Attention la méthode *setRotation()* attend en paramètre des radians alors que la méthode *setAngle()* attend un angle en degré.

Remarques :

- La fonction créée pour l'exercice ci-dessus est purement pédagogique car dans la pratique, nous utiliserons celle qui est fournie par Phaser : `this.rnd.between(min, max)`.
- On peut maintenant ajouter à notre template (et donc à nos projets) la fonction `init()` qui va permettre d'initialiser les variables globales du projets. Par exemple :

```
let config = {  
    ...,  
    scene: { init: init,  
        preload: preload,  
        create: create,  
        update : update }  
};  
  
let game = new Phaser.Game(config);  
let shipPositionX;  
  
function init() {  
    this.shipPositionX = 100;  
}  
...
```

Exercice R-Type :

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-Type⁶

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 (solution dans le fichier *ShipMove.html*)
 - Ajoutez dans la scènes des petites étoiles de manière aléatoire (solution dans le fichier *ShipMoveWithStars.html*)
 - Limitez les déplacements du vaisseau aux frontières de l'espace de jeu (solution dans le fichier *ShipMoveWithLimits.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*.

⁶ <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 Awakening⁷

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).

⁷ <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 = this.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(Phaser.Math.Between(0, 800),
                 Phaser.Math.Between(0, 600),
                 'star');
}
```

Grâce aux groupes, on peut également réaliser un pool d'objets (une réserve) dans lequel on viendra chercher des éléments pour construire le jeu. On peut, par exemple, créer une réserve de missiles pour lancer sur un vaisseau. Ce pool se déclare de la manière suivante :

```
let bullets = this.physics.add.group({
    defaultKey: 'star',
    maxSize: 20
});
```

Une fois cette réserve créée, il suffit de prélever un élément dans ce pool quand on en aura besoin :

```
let bullet = bullets.get();
if (bullet) {
    bullet.setPosition(groundEnnemy.x, groundEnnemy.y);
    ...
}
```

Vous remarquerez l'instruction *if(bullet)* qui vérifie si une « bullet » est bien disponible dans la réserve avant de la rendre active. Cela signifie que lorsqu'un objet disparaît du jeu (explose ou sort de l'écran) il faut le détruire pour qu'il soit à nouveau disponible dans la réserve.

Exercice :

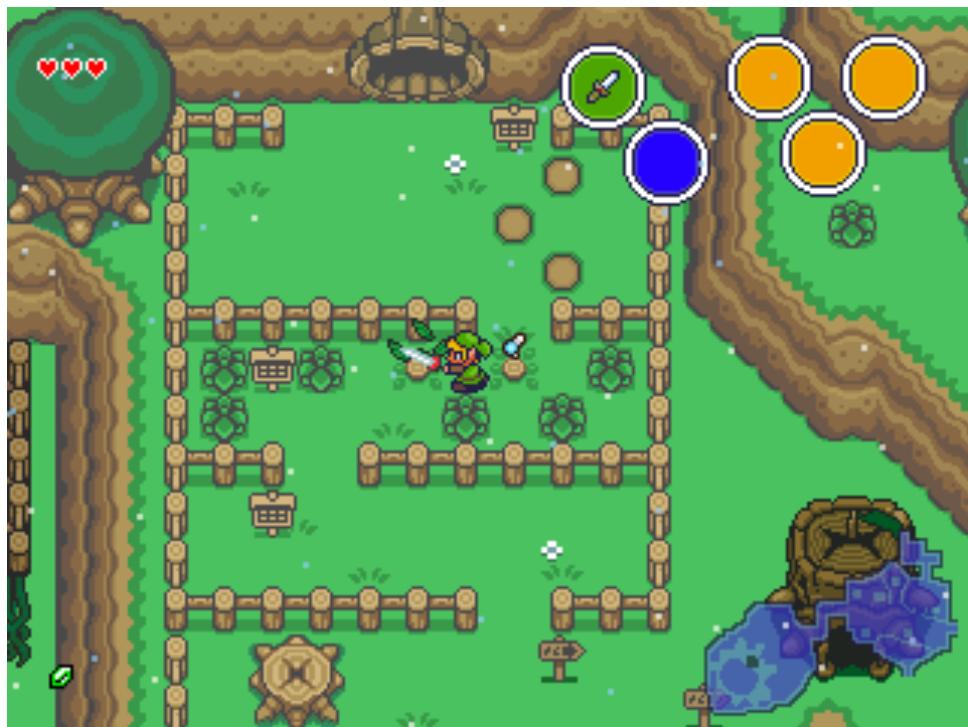
- Créez un groupe d'étoiles et faite-le se déplacer d'un seul bloc vers la gauche pour donner l'illusion d'un univers en mouvement (solution dans le fichier *ShipMoveWithMovingStars.html*).

Utilisez la fonction :

```
Phaser.Actions.IncX(stars.getChildren(), -1);
```

3.2.TileSet

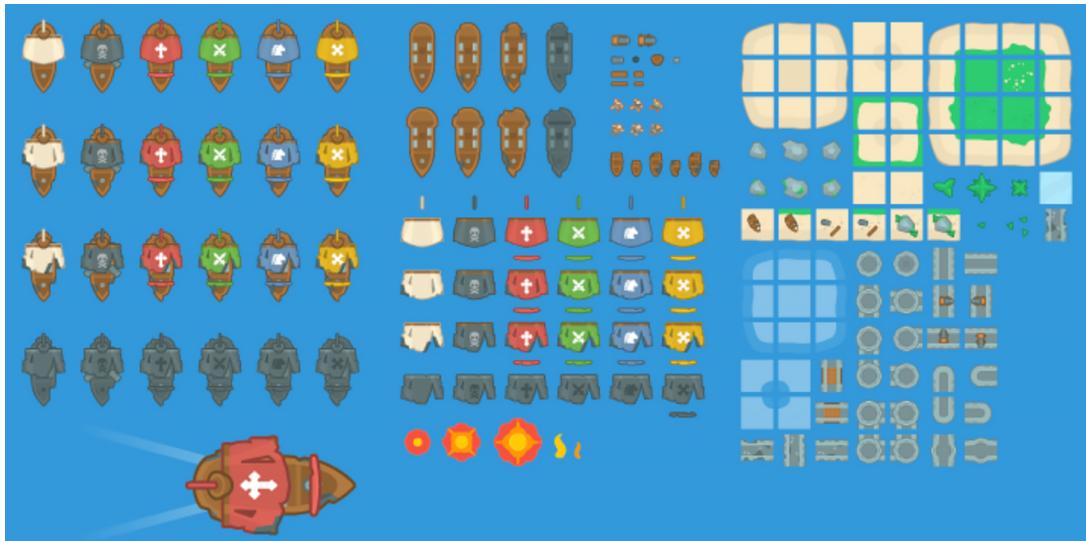
Pour créer le décors d'une *map*, on pourrait charger une seule image complète pour l'ensemble du niveau. Cette image serait alors considérée par *Phaser* comme un « gros bloc de pixels » avec lequel il serait impossible d'interagir (savoir si un élément est bloquant ou pas). Par exemple dans cet écran :



Exemple d'écran multi-layers

On doit pouvoir distinguer les barrière, de la prairie, de la montagne. On va donc créer différentes couches (*Layers*) à partir de tuiles (*Tiles*). C'est pour cela que dans les jeux vidéos en 2D, on utilise fréquemment des « *TileSet* » (ensemble de tuiles). Il s'agit en fait d'une image qui contient un ensemble d'images plus petites, souvent carrées et appelées « tuiles » ou *Tiles*. Assemblées judicieusement entre elles , elles forment un environnement de jeu.

Par exemple ce *TileSet* :



Exemple de TileSet⁸

Va permettre de réaliser un jeu tel que :



Exemple de jeu construit à parti du TileSet

⁸ <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');
```

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);
```

Nous avons déjà évoqué plus haut qu'il est intéressant de construire une *Map* en plusieurs couches (*layers*). Chaque couche représente un ensemble cohérent et représentatif du niveau. Par exemple, on peut créer un *layer* pour les décors « bloquant » (qui empêche un personnage de passer) et une couche pour les éléments sur lesquels le personnage peut circuler librement. Nous pouvons créer un *layer* 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);  
}
```

Enfin, il faut terminer cette suite d'instructions par un :

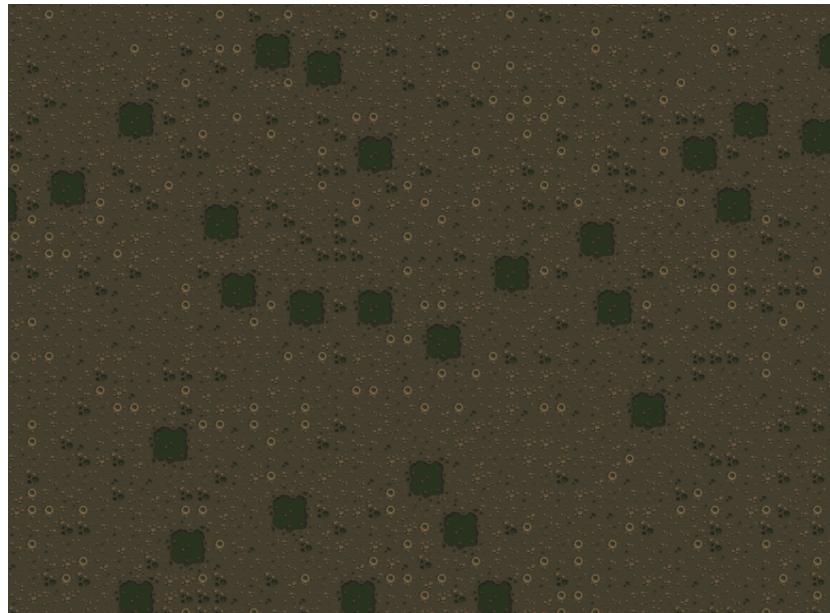
```
layer1.resizeWorld();
```

Exercices :

- A partir des différents codes ci-dessus créez un programme pour afficher le « *TileSet* » complet (*sci-fi-tiles.png* fourni dans les *Assets*) sur une ligne (ou deux). Il y a 56 tuiles de 16x16 pixels dans ce fichier et la première tuile est transparente. La solution est dans le fichier *DisplayTileSet.html*.
- Générez ensuite une *map* aléatoire complète en utilisant le même *TileSet*. La solution est dans le fichier *RandomMap.html*.

Exercice facultatif :

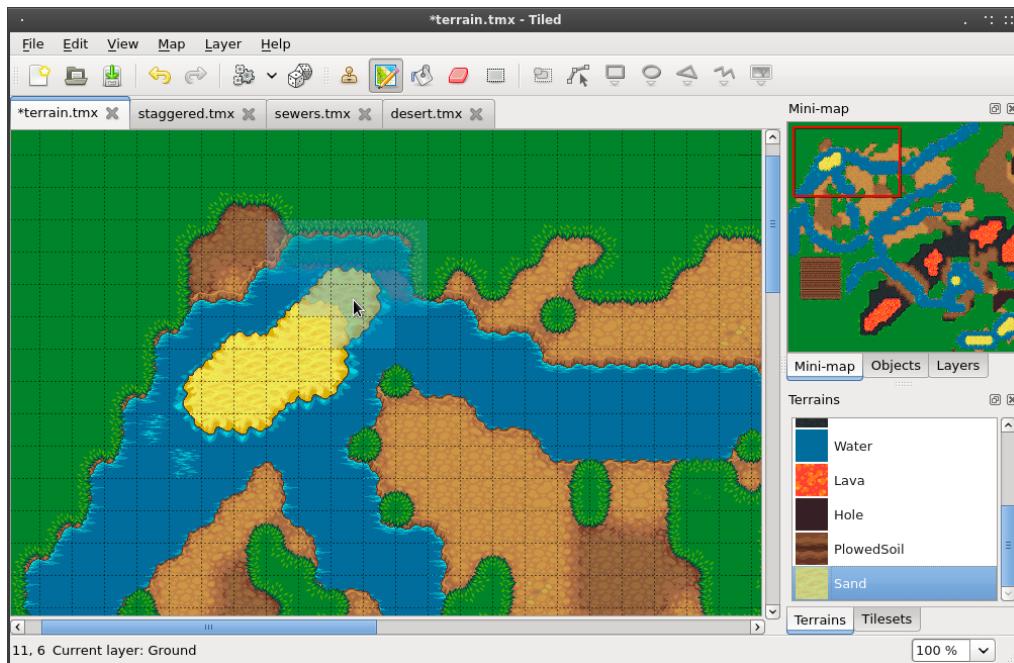
- Améliorez la génération aléatoire de la *map* avec des tuiles différentes et arrangées de manière cohérente, pour obtenir un résultat similaire à ce *screenshot* :



Génération aléatoire d'une map

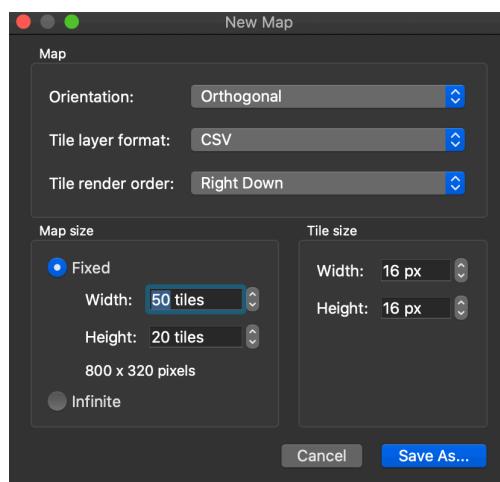
3.3.Tiled

Si vous voulez utiliser une *map* fixe (qui n'est pas créée aléatoirement au chargement du jeu), vous pouvez la créer avec un logiciel tel que *Tiled*⁹. C'est un logiciel qui permet d'éditer intuitivement un niveau de jeu à partir de tuiles à répartir dans une grille.



Screenshot du logiciel Tiled

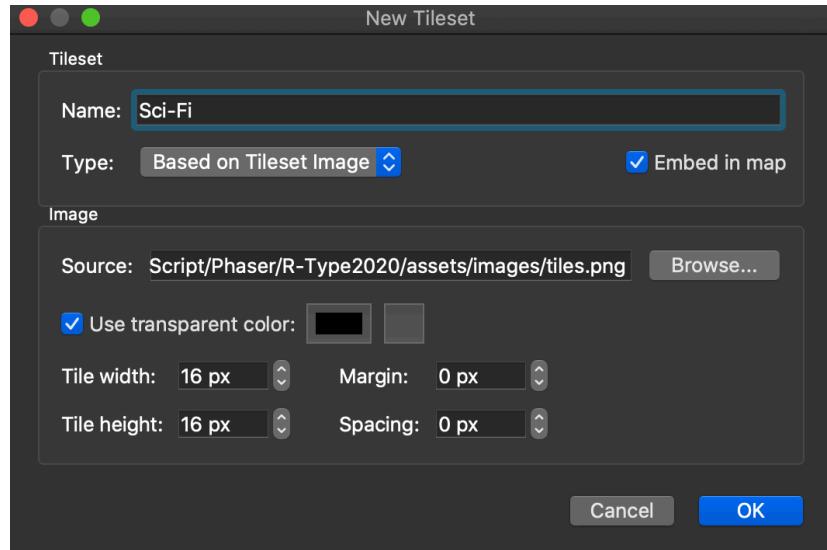
Lorsque vous lancez *Tiled*, vous devez créer une nouvelle carte (bouton « *New Map* »), en respectant les choix donnés dans la fenêtre ci-dessous (attention la largeur est de 150 *tiles* et la hauteur de 20) :



Création d'une map dans Tiled

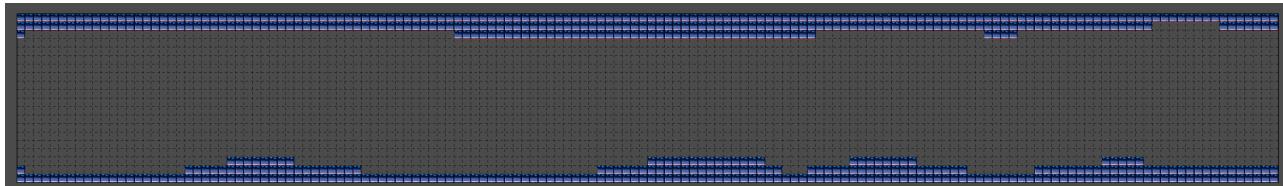
⁹ <http://www.mapeditor.org/>

Après avoir choisi un nom de sauvegarde pour votre fichier (extension *.tmx*), vous devez lui associer un *TileSet*, cliquez sur le bouton « *New Tileset* » :



Association d'un Tileset dans Tiled

Faites bien attention de cocher la case « *Embed in map* » et retenez bien le nom que vous donnez à votre *Tileset* (« *Sci-Fi* » dans notre exemple). Il faut ensuite « dessiner votre map dans la fenêtre principale de *Tiled* :



Dessin de la map dans Tiled

Attention, les tuiles que vous mettez à l'écran rentreront en collision avec votre vaisseau, il faut donc laisser un couloir vide au centre de l'écran pour qu'il puisse se mouvoir.

Vous pouvez finalement sauvegarder la *map* dans un format *json* (cliquez sur le menu *File->Export*), dont vous pouvez visualiser le contenu dans *Brackets* :

```
...
    "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":"Sci-Fi",
    "spacing":0,
    "tilecount":56,
...
]
```

Extrait du fichier JSON créé par le logiciel Tiled

Nous allons retourner dans *Brackets* pour utiliser notre *map* (au format *json*) dans *Phaser*. Faites bien attention au fait que les noms qui se trouvent ci-dessus dans les fichiers *JSON* (en couleur) doivent correspondre exactement aux paramètres passés dans les méthodes *Phaser* ci-dessous (de la même couleur).

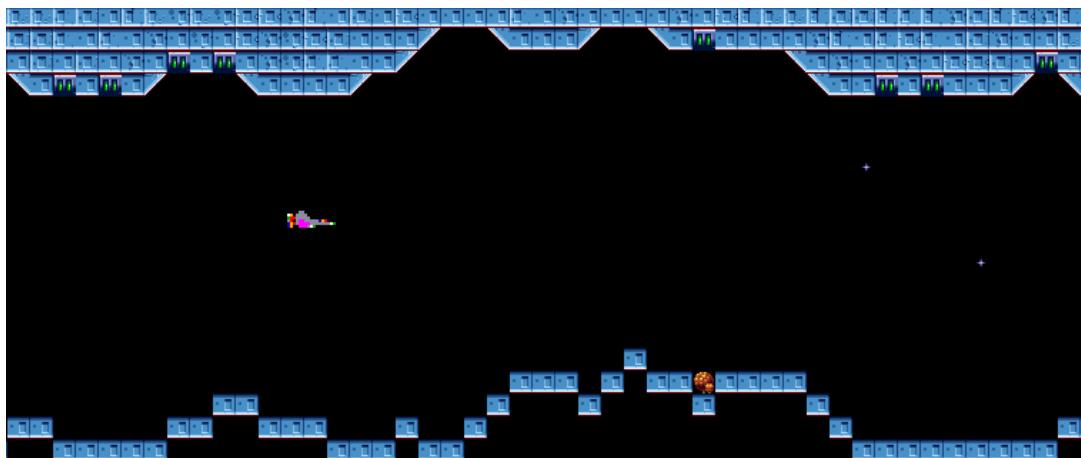
Le code dans votre programme est alors le suivant :

```
function preload() {  
    ...  
    this.load.image('tiles', './assets/images/tiles.png');  
    this.load.tilemapTiledJSON('backgroundMap',  
        './assets/tiled/newlevel.json'); }  
  
function create() {  
    const map = this.make.tilemap({ key: 'backgroundMap' });  
    var sciti = map.addTilesetImage('Sci-Fi', 'tiles', 16, 16, 0, 0);  
    var layer = map.createStaticLayer(0, sciti, 0, 0);
```

Vérifiez bien que dans la méthode `addTilesetImage()`, le premier paramètre (en bleu) a la même valeur que le *TileSet name* dans le fichier json

Exercice R-Type :

Créer avec le logiciel *Tiled* et à partir du *Tileset* donné (*tiles.png*¹⁰ dans les *Assets*), une *map* pour votre *R-Type*. Intégrez là dans votre application *R-Type* qui devrait alors ressembler à quelque chose de semblable (en fonction des tuiles utilisées) :



Exemple de map pour le R-Type

Pour plus d'informations voir l'article suivant : <https://medium.com/@michaelwesthadley/modular-game-worlds-in-phaser-3-tilemaps-1-958fc7e6bbd6>

¹⁰ <http://www.spriters-resource.com/resources/sheets/49/52083.png>

3.4.Tableaux

Nous venons de voir que la plupart des *maps* consistent en une grille (un ensemble de cases) qui contient des images (tuiles). On utilisera fréquemment les tableaux pour stocker les indices des *Tiles* dans la grille, pour par exemple, construire une map aléatoirement avant de l'afficher. 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

var inventory = ["axe", "water", "bread"];
// crée une variable inventory de 3 éléments de type string et
assigne les valeurs données entre crochets

points[12] = 6;
// assigne la valeur 6 au 13ème élément du tableau
```

En *JavaScript* les tableaux sont en fait des listes de valeur qui peuvent être manipulées efficacement avec des méthodes comme *push()*, *pop()*, *shift()*, ... Consultez la documentation pour plus d'information, quand vous en aurez besoin.

Exercice facultatif pour le R-Type :

Créez à partir du *Tileset* donné (*tiles.png*¹¹ dans les *Assets*), un environnement aléatoire mais cohérent que vous stockez dans un tableau avant de l'afficher. 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
 - avec la même tuile faites en bas de l'écran une « courbe » qui souligne un chemin
 - faites la même chose en haut de l'écran
 - vous devez maintenant gérer le fait que les deux « lignes » du dessus et dessous peuvent parfois s'entre-croiser
 - essayez de démarrer vos « lignes » ailleurs que dans les coins
 - adaptez la position initiale du vaisseau en fonction du chemin
 - 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
 - essayez de trouver dans le *TileSet* des tuiles qui peuvent s'enchaîner côte à côté et construisez une suite de tuiles cohérentes et aléatoires :
 - * créez un tableau de *tiles* qui contiendra la base de notre chemin
 - * « adoucissez » les angles en remplaçant les trous entre les blocs
 - * remplissez les espaces vides en haut et en bas de l'écran
 - positionnez un ennemi au sol de manière aléatoire mais conservez sa position en mémoire

¹¹ <http://www.spriters-resource.com/resources/sheets/49/52083.png>

4. Evénements temporels

Dans un jeu vidéo, il est souvent intéressant de générer des événements temporels (déclenchés à un certain moment ou à une certaine fréquence), pour par exemple, faire apparaître automatiquement des ennemis ou leur faire tirer des missiles de manières régulières.

4.1.Événements déclenchés après un délai

En *Javascript*, pour déclencher un événement (une fonction, par exemple) après un certain temps, il suffit d'utiliser la fonction *setTimeOut()* dont la syntaxe est la suivante :

```
setTimeout(myFunction, timeInMilliseconds)

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

Exercice :

Faites apparaître une étoile 5 secondes après que l'utilisateur aie appuyé sur la touche « flèche droite ». Solution dans le fichier *DisplayStarDelay.html*.

4.2.Événements générés en rafale

Si vous désirez générer un événement à une fréquence donnée, par exemple, pour faire tirer un canon de manière régulière ou pour faire apparaître des vaisseaux ennemis (*spawn*) par vague, vous pouvez utiliser la fonction polyvalente proposée par *Phaser* :

```
let timer = this.time.addEvent({
    delay: 500,                      // ms
    callback: myFunction,
    //args: [],
    callbackScope: this,
    repeat: 4
});

function myFunction() {
    // code to execute 4 times every 500 mSECONDS
}
```

Pour plus d'informations sur les *Timers*, référez-vous à la documentation ou sur ce site : <https://rexrainbow.github.io/phaser3-rex-notes/docs/site/timer/>

Exercice :

Reprenez l'exercice qui remplit l'écran d'étoiles mais générez ces étoiles une à une à la cadence d'une étoile toutes les deux secondes. Solution dans le fichier *SpawnStars.html*.

Exercice « R-Type » :

Nous avons créé une base ennemie, nous allons lui faire tirer des missiles. Créez un tir dans n'importe quelle direction, toutes les cinq secondes, depuis la station ennemie. Solution dans le fichier *.html*.

Nous voudrions maintenant faire tirer la base ennemie en direction du vaisseau du joueur. Nous avons besoin pour cela de quelques notions de physique et d'opération sur les vecteurs.

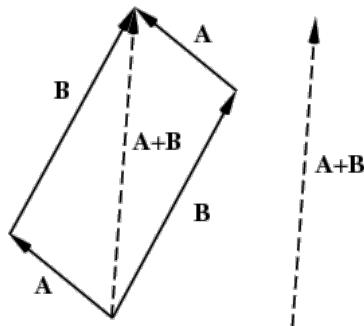
5. Physique

Nous allons aborder les notions liées à la physique dans *Phaser* : la vitesse des objets, l'application de la gravité et la détection des collisions.

5.1. Vitesse

Nous avons déjà vu plus haut comment donner une vitesse à un objet. Si nous voulons aller plus loin dans ce concept, nous avons besoin de notions sur les **vecteurs**.

5.1.1. Addition et soustraction de vecteurs



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

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

Exercice R-Type :

- A partir des notions de soustraction des vecteurs, faites tirer la base ennemie en direction du vaisseau (quelque soit sa position).

Vous remarquerez sans doute 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. 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 :

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

Exercice R-Type :

- Faites en sorte que la vitesse de tir soit constante quelque soit la position du vaisseau.

5.2. Collisions

Phaser offre une fonctionnalité très pratique : la détection de collisions. Vous n'êtes en effet pas obligé de calculer à chaque *frame* si votre vaisseau est entré en collision avec les décors ou un missile. *Phaser* peut effectuer cette vérification sans devoir écrire de code.

5.2.1. Collisions entre *sprites*

Pour détecter une collision entre deux *sprites* (ou un groupe), il suffit de vérifier si leurs surfaces se superposent ou pas. La méthode *getBounds()* permet d'obtenir les renseignements sur la surface (rectangle) occupée par un *sprite*. Il suffit alors de comparer si deux de ces « bounds » se superposent ou pas avec la méthode fournie par *Phaser* :

```
Phaser.Geom.Intersects.RectangleToRectangle(  
    this.sprite1.getBounds(),  
    this.sprite2.getBounds()  
) ;
```

Cette méthode renvoie un booléen qu'il suffit de vérifier avec un *if()*. Par exemple, pour redémarrer la scène quand une collision survient on fait :

```
if (Phaser.Geom.IntersectsRectangle(  
    this.sprite1.getBounds(),  
    this.sprite2.getBounds())  
) this.scene.restart();
```

Si on utilise le moteur physique, pour détecter une collision entre deux *sprites* (ou un groupe), il suffit de demander à *Phaser* de vérifier s'il y a collision dans la fonction *update()* avec la méthode :

```
this.physics.add.collider(player, bullets, collisionPlayerBullet,  
    null, this);
```

Les deux premiers paramètres de la fonction détermine les deux *sprites* incriminés dans la collision et le troisième paramètre spécifie le nom de la fonction qui est appelée quand une collision est détectée. Il faut donc ensuite définir la fonction en question :

```
function collisionPlayerBullet(_player, _bullet) {  
    _player.setVisible(false);  
    _bullet.destroy();  
}
```

Dans l'exemple ci-dessus la fonction détruit le missile et fait disparaître le vaisseau.

5.2.2. Collisions avec les *tiles*

Pour détecter les collisions avec les différents *layers* d'une map, il faut stipuler au moteur de jeu quels sont les *tiles* qui provoquent des collisions, avec la méthode *SetCollisionBetween()*. On spécifie comme paramètres l'intervalle des indices de tuiles incriminés dans les collisions. Par exemple, si on décide que les *tiles* d'indice 1 à 2055 provoquent des collisions, on écrit :

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

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

```
this.physics.add.collider(player, layer, collisionPlayerLayer,  
    null, this);
```

Lorsque *Phaser* détecte une collision entre les deux objets, il lance la fonction de *callback* : *collisionLayerSprite()* :

```
function collisionPlayerLayer(_player, _layer) {  
    _player.setVisible(false);  
}
```

Exercice R-Type :

Déetectez les collisions entre le vaisseau, les missiles et les décors.

5.3. Gravité

Dans notre jeu *R-Type* tout se passe dans l'espace, sans gravité. Si vous réalisez un jeu « sur une planète », vous pouvez ajouter à votre environnement de la gravité. Cette gravité peut être appliquée par défaut à l'ensemble des objets qui sont soumis au moteur physique :

```
this.physics.arcade.gravity.y = 100;  
this.physics.enable('star', Phaser.Physics.ARCADE);
```

Mais on peut aussi donner des valeurs de gravité différentes en fonction des objets. Pour cela on spécifie pour un objet particulier :

```
star.body.gravity.y = 200;
```

Si vous désirez faire rebondir votre objet, il faut d'abord spécifier que les frontières de l'écran « *collide* » (provoque des collisions) avec notre *sprite* :

```
star.body.collideWorldBounds = true;
```

Ensuite, il faut définir la valeur du rebond (l'élasticité) de l'objet :

```
star.body.bounce.y = 0.8;
```

Et si vous désirez momentanément supprimer la gravité sur un objet, il suffit de faire :

```
star.body.allowGravity = false;
```

6. Animations

Une animation en 2D peut s'envisager de différentes manières : avec un simple *Timer* (voir les événements temporels, plus haut), on peut remplacer l'image d'un *sprite* par une autre, on peut également utiliser des « *sprites* animés ». Par exemple, pour réaliser une explosion ou encore pour mettre une animation en boucle et donner une impression de mouvement, pour faire courir un personnage.

6.1. Avec *timer*

La plus simple animation que l'on puisse faire consiste à changer une image par une autre ou encore plus simplement de la faire disparaître, puis réapparaître (effet de clignotement). Par exemple, quand un vaisseau est touché, on peut le faire clignoter quelques secondes pour faire comprendre au joueur qu'il est touché et le rendre invincible temporairement pour laisser le temps au joueur de se repositionner.

Pour rendre un *sprite* invisible, il suffit de mettre son alpha (niveau de transparence au plus bas) :

```
star.alpha = 0.1;
```

Cette action doit être réalisée dans une fonction temporisée (vue plus haut).

Exercice R-Type :

Faites « clignoter » (afficher/cacher à une certaine fréquence) votre vaisseau pendant 3 secondes lorsqu'il est touché.

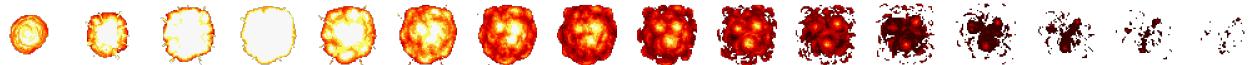
6.2. Avec *tween*

On peut faire la même chose de manière très simple grâce au système de *tweens* dans *Phaser* :

```
this.tweens.add({
    targets: [player],
    alpha: 0,
    ease: 'Linear',
    duration: 500,
});
```

6.3.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 **128 x 128**) :

```
function preload() {
    this.load.spritesheet('boom', '../Assets/explosion.png', {
        frameWidth: 128,
        frameHeight: 128
    });
}
```

Ensute il faut créer une animation basée sur ce *SpriteSheet* dans la fonction *Create()* :

```
function create() {
    var explosionAnimation = this.anims.create({
        key: 'explode',
        frames: this.anims.generateFrameNumbers('boom'),
        frameRate: 10,
        repeat: 0
    });
}
```

Et finalement dans la fonction de détection de collision, afficher le sprite et faire jouer l'animation avec un *play* :

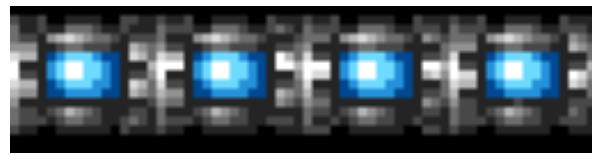
```
function ...() {
    var explosion = this.add.sprite(_player.x, _player.y, 'boom');
    explosion.play('explode');
}
```

Exercice R-Type :

Générez une animation qui montre une explosion du vaisseau lorsqu'il est touché par les décors ou un missile.

6.4.Animation bouclée

Contrairement au cas précédent dans lequel l'animation est lancée une seule fois (*one shot*) et disparaît, on peut faire boucler une animation. C'est à dire qu'à la dernière image du *spritesheet*, l'animation reprend en boucle au début et ainsi de suite infiniment.



Exemple d'animation bouclée dans un spritesheet

Pour cela, il suffit de donner une valeur de -1 au paramètre *repeat* :

```
function create() {  
    var explosionAnimation = this.anims.create({  
        key: 'explode',  
        frames: this.anims.generateFrameNumbers('spinner'),  
        frameRate: 10,  
        repeat: -1  
    });  
}
```

Exercice R-Type :

Envoyez des vaisseaux ennemis animés (fichier *bluemetal_32x32x4.png* dans les *assets*) de manière régulière.

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;  
}
```

Dans cet exemple, le constructeur est *Ennemy* et les deux propriétés sont *life* et *experience*.

7.2.Instanciation d'un objet

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

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

En *Javacript*, on peut définir un objet encore plus rapidement (sans créer de classe préalablement) en utilisant la syntaxe suivante :

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

7.3.Propriétés

Nous pourrons ensuite accéder aux propriétés des objets en y faisant référence d'une des manières suivantes :

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

7.4.Méthodes

Il y a deux manières de définir une méthode dans une classe : dans la méthode « constructeur » ou en utilisant les « prototypes ». 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.

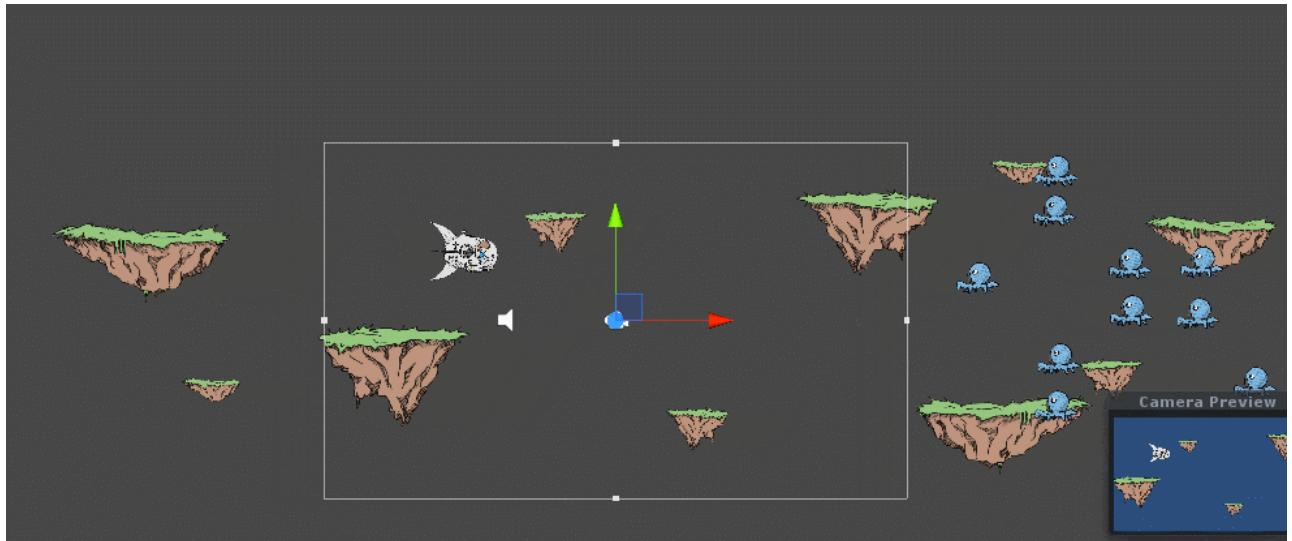
```
Person.prototype.hitted = function() {  
    this.life -- ;  
}
```

Exercice R-Type :

Le vaisseau est déjà un objet, ajoutez lui une nouvelle propriété qui va stocker le nombre de collisions avec un missile et le faire exploser après trois « touches ».

8. Camera

La caméra est un objet qui délimite une partie de l'aire de jeu (la *map* complète) et la visualise à l'écran. Votre *map* peut donc être plus grande que ce que la caméra « regarde ». Ceci va nous permettre, par exemple, de préparer différents écrans et de *switcher* rapidement de l'un à l'autre (changement de plan) mais aussi de balayer l'aire de jeu progressivement (*scrolling*).



*Illustration du scrolling par mouvement de caméra*¹²

Nous allons d'abord créer une aire de jeu qui soit plus grande que la résolution de la caméra, par exemple :

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

Dans ce code, nous créons une *map* qui fait **200 × 16 = 3200 pixels** de large, alors que la caméra du jeu ne fait que **800 pixels** de large et on peut alors effectuer un scrolling horizontal vers la droite en faisant simplement :

```
this.cameras.main.scrollX +=1;
```

Exercice R-Type :

Réalisez un niveau plus large que ce que la caméra « visualise » et faites *scroller* horizontalement la caméra. Adaptez l'apparition des ennemis au nouveau niveau.

¹² <http://pixelnest.io/tutorials/2d-game-unity/parallax-scrolling/>

9. Audio

On peut voir l'audio sous deux aspects : d'une part des sons ou musiques d'ambiance qui boucle sans fin et d'autre part les sons qui sont déclenchés suite à un événement (une explosion suite à un missile sur notre vaisseau). Il faut d'abord ajouter une ligne dans la configuration du jeu :

```
var config = {
    type: Phaser.AUTO,
    ...
    audio: {
        disableWebAudio: true
    }
};
```

Il faut ensuite, comme pour une insertion d'images dans la fonction *preload()* précharger le fichier avec le contenu audio :

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

Ensuite, dans la fonction *create()*, on instancie un objet qui va contenir le son et permettre sa gestion (jouer, mettre en pause, boucler le son, ...):

```
explosionSound = this.sound.add('explosionSound');
```

Pour créer un son suite à un événement (dans la fonction de *callback*, suite à une collision, par exemple), on va faire simplement :

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

Et si une musique doit « tourner » en boucle :

```
track.play('musicToLoop', {
    loop: true
});
```

Pour aller plus loin avec l'audio vous pouvez essayer de mettre en pause, de reprendre un morceau où vous l'avez arrêté, de changer le volume ou encore de le « *muter* », respectivement en utilisant les instructions suivantes :

```
ambiantSound.pause();  
ambiantSound.resume();  
ambiantSound.volume += 0.1;  
ambiantSound.mute = false;
```

Exercice R-Type :

Ajoutez une musique d'ambiance au jeu ainsi que des bruits de détonation lorsque les vaisseaux explosent.

10.Interface utilisateur

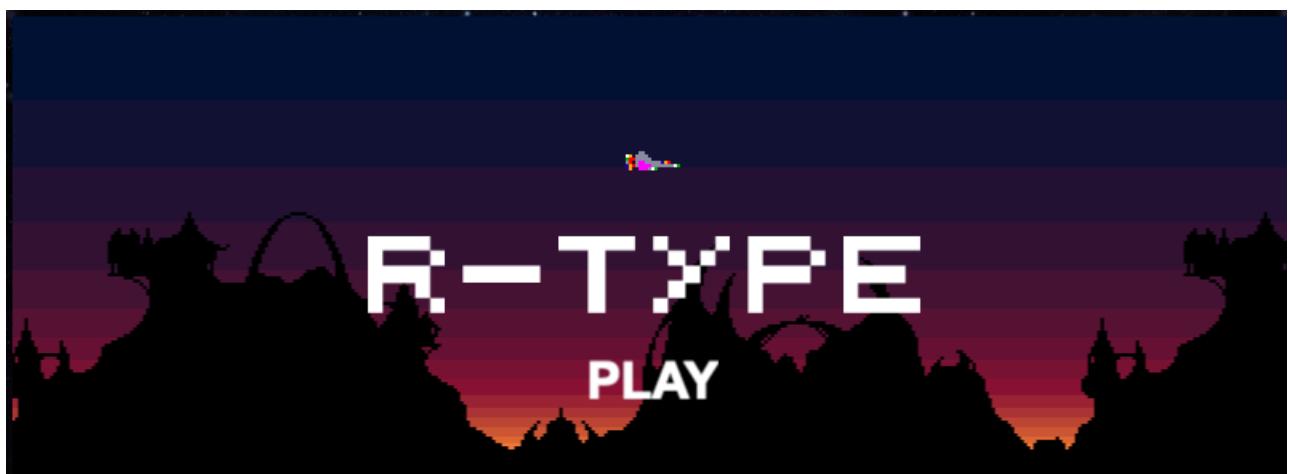
On ne peut pas imaginer un jeu dépourvu d'écrans d'accueil, de « Game Over » qui donne la possibilité de changer la configuration ou de rejouer.

10.1.Ecran d'accueil

Pour réaliser un écran d'accueil, c'est relativement simple, placer quelques images, un musique en boucle, un peu de texte et éventuellement une animation sur l'écran et vous avez votre écran de démarrage. Toutes ces choses ont été vues plus haut.

Exercice R-Type :

Ajoutez un écran d'accueil à votre jeu. Par exemple :



10.2.Interactivité

Dans l'écran que vous venez de créer, il serait évidemment intéressant de lancer le jeu quand l'utilisateur clique sur le bouton (image ou texte) « Play ». Il faut donc rendre cet élément interactif :

```
nomDeL'Element.setInteractive();
nomDeL'Element.on('pointerdown', function(pointer) {
    console.log("Element clicked");
});
```

Exercice R-Type :

Ajoutez de l'interactivité sur le texte « *play* » de l'écran d'accueil : lancez le jeu quand on clique sur le texte.

10.3. Informations « in game »

Il peut être intéressant d'afficher des informations à l'écran pendant le jeu, par exemple, le score, le nombre de vies, ... Il suffit pour cela de créer un texte à l'écran qui sera rafraîchi à chaque *frame*, dans la méthode *create()*.

Exercice R-Type :

Ajoutez un score en « surimpression » dans le jeu (le nombre de vaisseaux détruits, par exemple).

11. 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.

12. 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 :

```
this.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 - 2018</div>
```

13.Organisation du code

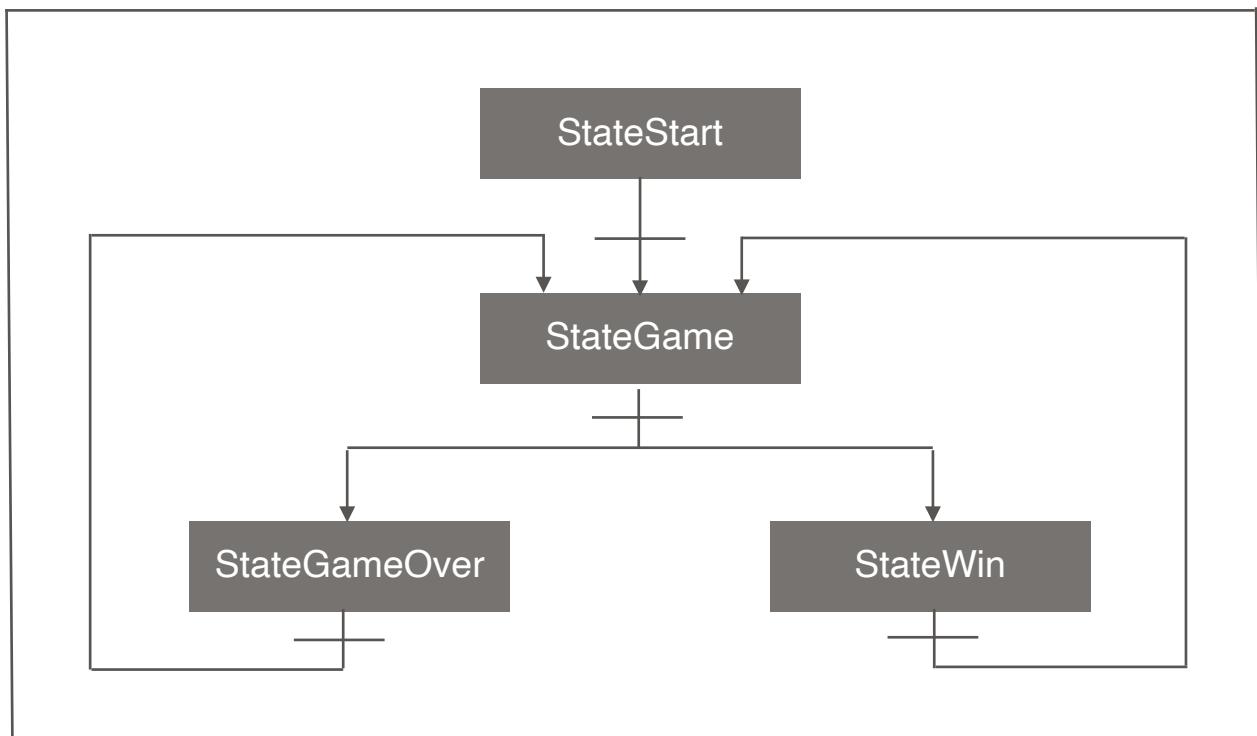
13.1.Machine à états finis

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



Machine à états de notre R-Type

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).

13.2. 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

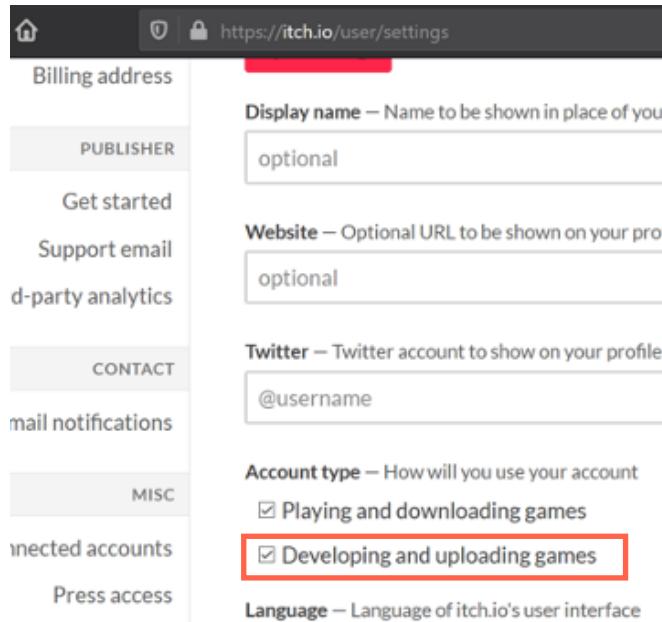
Ce projet s'étalera tout au long du cursus. Vous devez d'abord imaginer une application (ça ne doit pas nécessairement être un jeu) en 2D qui servirait de promotion à une marque, une entreprise, une organisation, une cause, ... Il pourrait s'agir d'une société pour la faire connaitre, d'un produit pour en faire la promotion, d'une théorie scientifique pour l'expliquer ... Ce projet ne doit donc pas être nécessairement commercial, vous pouvez mettre en lumière une association humanitaire, une ASBL, un centre de recherche,... Vous pouvez choisir un produit, une marque, une cause ou un principe qui vous parle et vous correspond. Quand le choix est fait, vous imaginez une application (idéalement ludique mais pas obligatoirement) qui dure 3 à 4 minutes maximum et qui, au final, promeut votre « projet ».

Quand votre idée est claire et précise, vous :

- Faites l'analyse sommaire 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 application et essayez de les planifier sur le nombre de jour dédiés au cursus.

14. Publier sur itch.io

D'abord vous « loger » sur itch.io, ou créer un compte si ça n'est pas encore fait. Vous assurer que votre compte est de type « développeur » (account type) :



Ensuite, cliquez « *Create New Project* » :

The screenshot shows the 'Creator Dashboard' with a 'Projects' tab selected. At the top, it displays '0 Views', '0 Downloads', and '1 Follower'. Below the tabs is a blue banner with a snowflake icon and the text 'Join Winter Sale 2020 – Put your projects on sale to participate Ends in 10 days' and a 'Create Sale with prefilled dates →' button. The main area shows a project card for 'Frogger' with a 'No Image' placeholder, the title 'Frogger', and a 'DRAFT' status. A red arrow points from the text 'Cliquez "Create New Project"' to the 'Create new project' button at the bottom of the card. To the right is a 'Summary' section with a 'View more →' link, a 'Views' chart (0 to 10), and a 'Downloads' chart (0 to 3). Below the summary is a section for 'Follow itch.io on Twitter and Facebook'.

Donnez-lui un nom, la classification : *game*, et un type de projet *HTML* (Kind of project) et choisissez « *\$0 or Donate* » ou « *No Payments* » dans le *Pricing* :

Title

Christmas Tree

Project URL

<https://rudigiotgmailcom.itch.io/christmas-tree>

Short description or tagline

Shown when we link to your project. Avoid duplicating your project's title

Optional

Classification

What are you uploading?

Games – A piece of software you can play

Kind of project

HTML – You have a ZIP or HTML file that will be played in the browser

TIP You can add additional downloadable files for any of the types above

Release status

Released – Project is complete, but might receive some updates

Pricing



\$0 or donate



Paid



No payments

Someone downloading your project will be asked for a donation before getting access.
They can skip to download for free.

Finalement, faites un *upload* du fichier *zip* (attention n'utilisez pas le *rar* mais bien le *zip*) avec le fichier *index.html*, les répertoires *js* et *assets*. Cochez la case « *Enable Scollbars* » si la taille de votre application le nécessite.

Uploads

Upload a **ZIP** file containing your game. There must be an `index.html` file in the **ZIP**.
Or upload a `.html` file that contains your entire game. [Learn more →](#)

Any additional files you upload will be made available for download. You can apply a minimum price to the project after uploading additional downloadable files.

ChristmasTree.zip 16mb · Change display name <input checked="" type="checkbox"/> This file will be played in the browser	More... Delete file
---	-------------------------------------

TIP Use butler to upload game files: it only uploads what's changed, generates patches for the [itch.io app](#), and you can automate it. [Get started!](#)

Upload files

or  [Choose from Dropbox](#)

[Add External file](#) (?)

File size limit: 1 GB. [Contact us](#) if you need more space

Embed options

How should your project be run in your page?

[Click to launch in fullscreen](#) ▾

Frame options

Mobile friendly — Your project can run on mobile phones (smaller resolution and touch support)

Enable scrollbars — Enable scrollbars in the iframe that contains your project

Attention, avant de faire votre fichier compressé, vérifiez que vos liens sont bien relatifs (du style `./assets/sprites/...`).

15. Atelier de Noël

Nous allons réaliser dans cet exercice un décor de Noël interactif. Vous aurez besoin des assets fournis en annexe.

15.1. Introduction

Il faut d'abord s'initier aux bases de *Phaser* et lire les chapitres jusqu'au 2.5 de ce document.

15.2. Images

Nous allons ensuite placer une image à l'écran et créer un écran pour obtenir le résultat suivant :



Dans la fonction *preload()* :

```
this.load.image('background', './assets/images/back_2.png');
```

Dans la fonction *create()* :

```
backImage = this.add.image(0, 0, 'background');
backImage.setOrigin(0, 0);
backImage.setScale(0.5);
```

Placez ensuite le sapin (tree_2.png), les guirlandes (ribbon.png) et quelques boules dans le décor (attention à l'ordre des images).

15.3. Boucle et nombres aléatoires

Ajoutez des étoiles (utilisez une boucle) dans le ciel de manière aléatoire. *Phaser* fournit une fonction très pratique :

```
Phaser.Math.Between(min, max);
```

Et pour rappel, en *JavaScript* une boucle peut s'écrire :

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

15.4. Fonction update() et transparence

Sachant que vous pouvez régler la transparence d'une image en modifiant sa propriété *alpha* (comprise entre 0 et 1), essayer de placer une étoile qui clignote.

```
blinkingStar.alpha = 0.5;
```

Essayez ensuite de la déplacer aléatoirement à chaque fois qu'elle s'éteint.

```
blinkingStar.setPosition(x, y);
```



15.5.Tween

On peut réaliser assez simplement des animations (comme sur l'étoile, précédemment) grâce au *tweens*. Par exemple, vous pouvez créer une guirlande (*ribbon*) plus claire et la superposer à celle qui est déjà à l'écran et lui appliquer un *tween* (variation programmée) de sa propriété *alpha* de la manière suivante :

```
let tweenRibbon = this.tweens.add({
    targets: ribbonClear,
    alpha: 1,
    duration: 2000,
    ease: 'Power2',
    yoyo: true,
    loop: -1
});
```

15.6.Pool d'objets

Vous allez maintenant ajouter de la neige qui tombe, il faut d'abord créer un flocon (dans un fichier *png*) et créer un groupe d'objet (un ensemble de flocons dans notre exemple) :

```
snowflakes = this.physics.add.group({
    defaultKey: 'snowflake',
    maxSize: 20
});
```

Ensuite, on va « puiser » dans notre réserve de flocons pour en afficher un à l'écran :

```
let snowflake = snowflakes.get();
if (snowflake) {
    snowflake.setPosition(Phaser.Math.Between(0, 611), 0);
}
```

Vous remarquerez l'instruction *if(snowflake)* qui vérifie si un « flocon » est bien disponible dans la réserve avant de l'afficher. Cela signifie que lorsqu'un objet disparaît

(sort de l'écran) il faudra le détruire pour qu'il soit à nouveau disponible dans la réserve et pouvoir réapparaître en haut de l'écran.

Vous remarquerez également l'instruction `this.physics.add()` qui va nous permettre d'ajouter de la physique à notre objet (dans notre exemple, une vitesse vers le bas, pour donner l'impression des flocons qui tombent du ciel) :

```
snowflake.setVelocity(0, 25);
```

Ensuite, vous pouvez, avec un « timer », faire apparaître un flocon de manière régulière en haut de l'écran (position horizontale et vitesse aléatoires). Basez-vous sur l'exemple de code suivant :

```
var timer = this.time.addEvent({
    delay: 500,                      // ms
    callback: myFunction,
    //args: [],
    callbackScope: this,
    repeat: 4
});

function myFunction() {
    // code to execute 4 times every 500 mSECONDS }
```

Il faut finalement détecter le moment où un flocon atteint le bas de l'écran pour le détruire (méthode `destroy()`) et permettre ainsi qu'il puisse être à nouveau libre dans le groupe et donc être réutilisé pour apparaître à nouveau :

```
snowflakes.getChildren().forEach(
    function(snowflake) {
        if (snowflake.y>980) snowflake.destroy();
    }, this);
```

15.7.Jouer une musique

Pour l'audio voir comment faire page 55. Choisissez une musique de Noël libre de droit (« royalty free ») et ajouter là à votre animation.

15.8.Boutons interactifs

Pour utiliser une image interactive (clickable), on peut utiliser un code similaire à ce qui suit :

```
snowButton = this.add.sprite(10, 10, 'snowButton').setInteractive();
snowButton.on('pointerdown', snowButtonDown);
```

où *snowButtonDown* est une fonction que vous devez définir :

```
function snowButtonDown() {
    ...
}
```

Vous pouvez, dès lors, créer trois boutons (images) qui permettent d'activer/désactiver la musique, de d'allumer/éteindre les lampes du sapin et de faire (ou pas) neiger.

15.9.Textes

Finalement, vous pouvez ajouter quelques emballages au pied du sapin et afficher une promotion aléatoire (par exemple, un code de réduction pour un achat sur le site) quand on clique sur un des cadeaux.

```
let winText = this.add.text(190, 142, 'text',
    { fontFamily: 'Arial', fontSize: 18, color: '#00ff00' });
```

15.10.Personnalisation

Vous pouvez, pour terminer, personnaliser votre carte de Noël, avec des images autres que celles que je vous ai donné, destinée à un site ou une institution de votre choix.

15.11.Publication sur Internet

Vous pouvez aussi publier (pour votre dossier portfolio) sur *Internet* votre application (sur [itch.io](#), par exemple).

16. Atelier Quiz

Nous allons réaliser dans cet exercice un Quiz. Vous aurez besoin des *assets* fournis en annexe.

16.1. Introduction

Il faut pour réaliser cet exercice avoir réalisé l'atelier de Noël ou alors d'abord s'initier aux bases de *Phaser* et lire les chapitres jusqu'au 2.5 de ce document.

16.2. Images

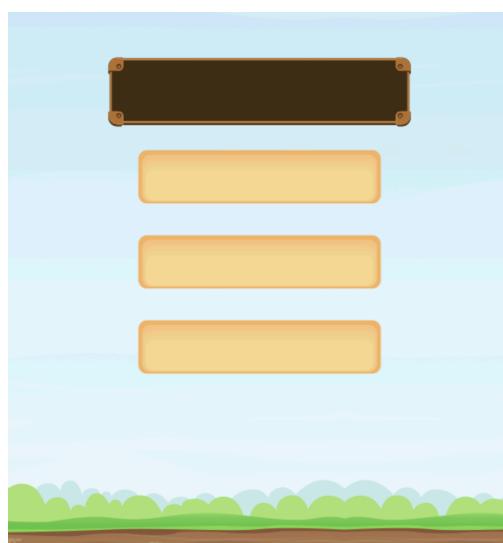
Nous allons d'abord mettre une image de fond pour notre Quiz. Pour rappel, il faut d'abord « pré-charger » l'image dans la fonction *preload()* :

```
this.load.image('background', './assets/images/back_2.png');
```

et ensuite dans la fonction *create()* :

```
backImage = this.add.image(0, 0, 'background');
backImage.setOrigin(0, 0);
backImage.setScale(0.5);
```

De la même manière, vous pouvez maintenant positionner le panneau pour la question (*Label1.png*) et trois panneaux pour les trois réponses possibles (*Label4.png*), pour obtenir un écran similaire à celui-ci :



16.3.Textes

Nous allons ajouter le texte à l'intérieur du panneau réservé à la question et les trois réponses possibles dans les trois autres panneaux. Pour rappel, on peut positionner un texte de cette manière :

```
myText = this.add.text(190, 142, 'texte \n avec retour à la ligne',  
{ fontFamily: 'Arial', fontSize: 18, color: '#00ff00' });
```

Vous pouvez utiliser les question et réponses d'un Quiz existant, par exemple :

<https://www.laculturegenerale.com/test-de-culture-generale-1/>



Ecran du Quiz avec la première question

16.4.Boutons interactifs

Pour rappel, si vous voulez utiliser une image « clickable », il faut la rendre interactive avec un code similaire :

```
myButton = this.add.image(10, 10, 'myButton').setInteractive();  
myButton.on('pointerdown', () => {answer(3)});
```

où *answer* est une fonction que vous devez définir avec un paramètre qui récupère le numéro de la question cliquée :

```
function answer(i) {  
    console.log(i);}
```

Vous pouvez, dès lors dans cette fonction tester si la réponse cliquée est la bonne et la mettre en vert et les autres (les fausses) en rouge. Vous pouvez aussi afficher une petite étoile dans le bas de l'écran en cas de bonne réponse et un bouton pour passer à la question suivante.

Pour rappel, vous pouvez changer la couleur et l'opacité des images et des textes avec ces instructions :

```
starImage.alpha = 0.3;  
starImage.tint = 0xff0000;  
answerText.setTextColor("#00AA00");
```

Vous aurez certainement besoin de temporairement désactiver l'interactivité d'un objet, vous pouvez alors utiliser :

```
objet.disableInteractive();  
  
// et pour réactiver l'interactivité :  
objet.setInteractive();
```

Vous aurez aussi sans doute besoin de temporairement rendre un objet invisible, ce qui peut se faire en le positionnant en dehors de l'écran, en mettant sa propriété alpha à zéro ou encore en utilisant :

```
objet.setVisible(false);  
  
// et pour le rendre visible à nouveau :  
objet.setVisible(true);
```

16.5.Physique

Pour rappel, une image que vous voulez soumettre à la physique (pour lui soumettre la gravité ou lui donner une vitesse) vous pouvez créer l'image en faisant :

```
starImage = this.physics.add.image(60, 600, 'star');  
// et lui donner une vitesse avec :  
starImage.setVelocity(0,0);
```

16.6. Objets JSON

L'idée est d'avoir un *Quiz* avec plusieurs questions. Nous allons donc stocker les questions et réponses dans un fichier JSON que nous chargerons dans un objet JavaScript. Voici la structure proposée pour l'arborescence du questionnaire :

```
- questions []
  - title
  - answers []
  - goodAnswer
```

Essayez de créer le fichier JSON correspondant à cette structure avec une dizaine de questions. Vous pouvez vous aider de ce site pour réaliser votre fichier : https://www.w3schools.com/js/js_json_arrays.asp. Chargez ensuite ce fichier dans une variable de votre code :

```
// dans le preload
this.load.json('questions', './assets/data/Questions.json');

//dans le create
questionJSON = this.cache.json.get('questions');

// pour utiliser un élément
console.log(questionJSON.questions[0].goodAnswer)
```

16.7. Tableaux d'objets

Pour les étoiles et les panneaux de réponses, vous avez intérêt à en faire des tableaux d'objets. C'est plus simple pour les gérer, on ne doit pas répéter les lignes de code identiques. On peut créer un tableau d'objets de la manière suivante :

```
let starImage = [];
for (let i = 0; i < 10; i++) {
    starImage[i] = this.add.image(30 + i * 60, 600, 'star');
    starImage[i].setScale(0.3);
    starImage[i].alpha = 0.3;
    starImage[i].setVisible(false);
}
```

16.8.Tween

On peut réaliser des animations grâce au *tweens*. Par exemple, vous pouvez créer un déplacement d'objet en appliquant un *tween* (variation programmée) de sa propriété *x* de la manière suivante :

```
let tweenPanel = this.tweens.add({
    targets: panel,
    y: 100,
    scaleY: 1.2,
    angle: 15,
    duration: 1000,
    ease: 'Power2',
    yoyo : true,
    loop: 0,
    paused: true
});
```

Vous remarquerez le dernier paramètre qui met le *tween* en pause à sa création, pour l'activer, il faudra utiliser l'instruction :

```
tweenPanel.play();
```

16.9.Son

Vous pouvez télécharger quelques sons qui vous plaisent, par exemple, à cette adresse : <https://mixkit.co/free-sound-effects/game/> et les insérer dans votre jeu. Pour rappel :

```
// dans le preload
this.load.audio('goodSound', './assets/Sound/good.wav');

// dans le create
goodAnswerSound = this.sound.add('goodSound');

// quand vous voulez jouer le son
goodAnswerSound.play();
```

16.10.Timer

Pour rappel, pour « retarder » l'exécution d'une fonction on peut utiliser un *timer* :

```
let timer = this.scene.time.addEvent({  
    delay: 1000,  
    callback: fonctionRetardee,  
    callbackScope: this  
});
```

16.11. Font personnelle

Pour utiliser une *font* personnelle (fichier *ttf*, par exemple) il faut utiliser un *loader Javascript*, une fonction dans le genre :

```
function loadFont(name, url) {  
    var newFont = new FontFace(name, `url(${url})`);  
    newFont.load().then(function (loaded) {  
        document.fonts.add(loaded);  
    }).catch(function (error) {  
        return error;  
    });  
}
```

On peut ensuite utiliser cette fonction de la manière suivante, dans le *create()* :

```
loadFont("carterone", "./assets/Fonts/CarterOne.ttf");
```

et l'utiliser dans la création d'un texte :

```
startText = this.add.text(150, 450, "Pressez sur le bouton", {  
    fontFamily: 'carterone',  
    fontSize: 18,  
    color: '#333333'  
});
```

17. Atelier Frogger

Nous allons réaliser dans cet exercice un jeu « old-school » qui s'appelle Frogger. Vous aurez besoin des *assets* fournis en annexe.



Le jeu Frogger original

17.1. Introduction

Il faut idéalement pour réaliser cet exercice avoir réalisé les deux ateliers précédents ou alors d'abord s'initier aux bases de *Phaser* et lire les chapitres jusqu'au 2.5 de ce document.

17.2. Images

Nous allons d'abord mettre une image de fond (background) pour notre jeu. Pour rappel, il faut d'abord « pré-charger » l'image dans la fonction *preload()* :

```
this.load.image('background', './assets/images/back.png');
```

et ensuite dans la fonction *create()* :

```
backImage = this.add.image(0, 0, 'background');
backImage.setOrigin(0, 0);
backImage.setScale(0.5);
```

De la même manière, vous pouvez ensuite afficher la petite grenouille en bas de l'écran, prête à traverser la route ainsi que sa maman qui l'attend de l'autre côté de la route (position horizontale aléatoire (attention : toujours positionner dans des multiples de 16 pixels). Pour ce faire, n'oubliez pas la fonction fournie par *Phaser* :

```
Phaser.Math.Between(min, max);
```

17.3.Interactivité clavier

Pour « capturer » les touches du clavier, nous pouvons dans le `create()`, assigner quatre variables (déclarées globalement) :

```
down = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.DOWN);
up = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.UP);
left = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.LEFT);
right = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.RIGHT);
```

Dans la fonction `update()` on va alors vérifier si ces touches sont enfoncées avec un :

```
if (Phaser.Input.Keyboard.JustDown(down)) ...
```

A partir de ces informations vous pouvez déplacer la grenouille dans les 4 directions (16 pixels à la fois) avec une instruction du style :

```
playerFrog.x += 16;
```

Vous pouvez ensuite « orienter » la grenouille dans le sens de son avancée avec :

```
playerFrog.setAngle(90); // l'angle est donné en degré
```

Finalement vous devez interdire à la grenouille de sortir de l'écran.

17.4.Détection de collisions

Phaser fournit plusieurs méthodes pour détecter des collisions entre *sprites*. Nous pouvons, par exemple, détecter dans la fonction *update()* si deux images se chevauchent (ont une intersection), avec les fonctions :

```
if (Phaser.Geom.Intersects.RectangleToRectangle (
    playerFrog.getBounds () , mumFrog.getBounds ()
))
```

Vous pouvez, dès lors détecter si la petite grenouille retrouve sa maman et afficher un joli coeur (qui grandit) pendant deux secondes et revenir ensuite à l'écran d'accueil avec la fonction :

```
this.scene.restart();
```

17.5.Moteur physique

Phaser peut gérer la physique des objets (vitesse, gravité, ...). L'image insérée doit donc répondre aux lois de la physiques, ajouter alors les images de la manière suivante :

```
car = this.physics.add.image(100, 100, 'carImage');
```

Vous pourrez alors donner une certaine vitesse à votre objet :

```
car.setVelocity(100, 0);
```

Avec ces instructions, placez une voiture à l'écran et donnez lui une vitesse horizontale. Détectez ensuite les collisions entre la voiture et la grenouille. Si la grenouille se fait écraser, changer son image et redémarrer la scène 4 secondes plus tard (voir comment définir un *Timer* page 71). Multipliez ensuite le nombre de voitures dans les deux sens de circulations (2 x 3 rangées de voitures). N'oubliez pas, pour ce faire, de créer un tableau de voiture :

```
let cars = []
```

Et d'utiliser la méthode *push()* pour ajouter un élément dans un tableau :

```
cars.push(car);
```

17.6.Son

Vous pouvez utiliser les sons fournis dans les assets pour les lire quand la grenouille bouge, quand elle se fait écraser ou qu'elle retrouve sa maman. Vous pouvez aussi ajouter un son d'ambiance de type trafic routier. Pour rappel :

```
// dans le preload  
this.load.audio('jump', './assets/Sound/coaac.wav');  
  
// dans le create  
jumpSound = this.sound.add('jump');  
  
// quand vous voulez jouer le son  
jumpSound.play();  
  
// quand vous voulez jouer un son en boucle  
traficSound.play({  
    loop: true  
});
```

17.7.Bouton interactif

Vous pouvez pour terminer, réaliser un écran d'accueil avec un bouton interactif qui permet de démarrer le jeu. Pour rappel dans le *create()* :

```
playButton = this.add.image(240, 290, 'play').setInteractive();  
playButton.on('pointerdown', startGame);
```

N'oubliez pas aussi de définir la fonction *startGame()* :

```
function startGame() {  
    introScreen.setVisible(false);  
    playButton.setVisible(false);  
    traficSound.play();  
}
```

18. Atelier R-Type

Nous allons réaliser dans cet exercice un jeu « old-school » qui s'appelle Frogger. Vous aurez besoin des *assets* fournis en annexe.



Le jeu R-Type original

18.1. Introduction

Il faut idéalement pour réaliser cet exercice avoir réalisé les ateliers précédents ou alors d'abord s'initier aux bases de *Phaser* et lire les chapitres jusqu'au 2.5 de ce document.

18.2. Images

Nous allons d'abord mettre une l'image du vaisseau du joueur pour notre jeu. Pour rappel, il faut d'abord « pré-charger » l'image dans la fonction *preload()* :

```
this.load.image('player', './assets/images/ship.png');
```

et ensuite dans la fonction *create()* ajoutez le vaisseau au moteur physique (on lui donnera plus tard une vitesse, pour le déplacer) :

```
playerShip = this.physics.add.image(0, 0, 'player');
```

18.3.Interaction clavier

Pour capturer les touches du clavier, nous avons déjà vu une méthode (plus haut) mais il existe une alternative qui est la suivante, dans la fonction *update()* :

```
cursors = this.input.keyboard.createCursorKeys();  
  
if (cursors.right.isDown) playerShip.setVelocityX(shipSpeed);
```

Répétez la même ligne pour les autres directions et limitez les déplacement du vaisseau à la zone de jeu. Créez ensuite un ennemi qui apparaît par la droite de l'écran, si il sort à gauche, il réapparaît à droite.

18.4.Collisions

Vous allez maintenant détecter si le joueur touche l'ennemi et alors faire redémarrer le jeu, pour rappel :

```
this.scene.restart();
```

On a vu plus haut qu'on peut détecter une collision avec les instructions :

```
if (Phaser.Geom.Intersects.RectangleToRectangle(  
    ennemyShip.getBounds(),  
    playerShip.getBounds())) {
```

Ajoutez la possibilité pour le joueur de tirer des missiles (horizontalement) en utilisant la barre d'espacement comme gâchette. Dans *create()* :

```
spacebar =  
this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.SPACE);
```

Et dans le *update()* :

```
if (Phaser.Input.Keyboard.JustDown(spacebar)) {  
    ...  
}
```

Créez un réservoir (group) de missiles, comme nous l'avons déjà vu plus haut. Pour rappel, pour créer le groupe d'images :

```
missiles = this.physics.add.group({  
    defaultKey: 'missile',  
    maxSize: 50  
});
```

Et pour « puiser » dans le réservoir :

```
let missile = missiles.get();  
if (missile) {  
    ...  
}
```

Pour détecter la collision entre le missile et le vaisseau ennemi nous allons utiliser une autre technique (*collider*) qui consiste à demander au moteur physique de détecter la collision entre deux éléments et d'appeler une fonction (dite de *callback*) si c'est le cas :

```
this.physics.add.collider(ennemyShip, missiles, collisionEnnemy,  
    null, this);
```

Il faut ensuite définir la fonction appelée en cas de collision :

```
function collisionEnnemy(ennemy, missile) {  
    missile.destroy();  
    ...  
}
```

Vous remarquerez au passage dans le code précédent comment détruire un objet avec la méthode *destroy()*.

Une autre méthode pour « faire disparaître » un objet est de le rendre invisible et de désactiver la physique (pour ne plus avoir de collision avec cet objet) :

```
player.setVisible(false);  
player.body.enable = false;
```

18.5.Timer

De la même manière, créez un ennemi au sol qui tire régulièrement des balles dans une direction aléatoire avec une détection de collision (collider). Faites disparaître le vaisseau quand il est touché trois fois par une de ces balles. Pour rappel, pour créer un timer, on peut faire :

```
var timerBulletShoot = this.time.addEvent({
    delay: 1000,
    callback: shootBullet,
    callbackScope: this,
    repeat: 20
});
```

N'oubliez pas de créer et définir la fonction *shootBullet()*.

18.6.Images animées

Vous allez maintenant afficher l'animation d'une explosion lorsqu'un objet explose (ennemi ou joueur). Dans la fonction *preload()* :

```
this.load.spritesheet('exAnim', './assets/animations/explosion.png',
{    frameWidth: 128,
    frameHeight: 128 });
```

Ensuite dans la fonction *create()*, il faut créer l'animation :

```
let explosionAnimation = this.anims.create({
    key: 'explode',
    frames: this.anims.generateFrameNumbers('exAnim'),
    frameRate: 20,
    repeat: 0,
    hideOnComplete: true
});
```

Et pour jouer l'animation :

```
let explosionAnim = this.add.sprite(playerShip.x, playerShip.y,
                                         'explosionAnim');
explosionAnim.play('explode');
```

18.7.Son

Pour ajouter du son, il faut simplement dans la fonction preload() :

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

Et au moment de lire le fichier son :

```
let explosionSound = this.sound.add('explosionSound');
explosionSound.play();
```

18.8.Tileset

Pour utiliser un jeu de tuiles (*Tileset*) reportez-vous au chapitre 3.2 concernant l'utilisation du logiciel *Tiled* et ensuite dans le *preload()* :

```
this.load.image('tiles', './assets/images/tiles.png');
this.load.tilemapTiledJSON('backgroundMap',
    './assets/tiled/newlevel.json');
```

dans le *create()* :

```
let map = this.make.tilemap({ key: 'backgroundMap' });
let sciti = map.addTilesetImage('Sci-Fi', 'tiles', 16, 16, 0, 0);
let layer = map.createStaticLayer(0, sciti, 0, 0);
```

il faut également dans le *create()* spécifier quelles tuiles rentrent en collision avec l'instruction (55000 représente toute les tuiles du *tileset*) :

```
layer.setCollisionBetween(1, 55000);
```

N'oubliez pas aussi de détecter les collision de notre vaisseau avec le décor :

```
this.physics.add.collider(playerShip, layer, collisionPlayer,
    null, this);
```

Comme le niveau est plus large que la surface de jeu, il faut faire scroller le décor avec dans l'*update()* :

```
if (this.cameras.main.scrollX < 2400) this.cameras.main.scrollX += 1;
```

18.9. Orientation d'un tir

Pour faire tirer l'ennemi au sol vers le vaisseau vous pouvez utiliser le code suivant :

```
let distance = Math.sqrt((playerShip.x - ennemyGround.x) ** 2 +  
                         (playerShip.y - ennemyGround.y) ** 2);  
let bulletSpeedX = (playerShip.x - ennemyGround.x) *  
                    bulletSpeed / distance;  
let bulletSpeedY = (playerShip.y - ennemyGround.y) *  
                    bulletSpeed / distance;  
bullet.setVelocity(bulletSpeedX, bulletSpeedY);
```