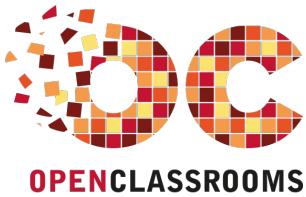
Créer un mini-RPG en JavaScript avec canvas

Par Sébastien CAPARROS (sebcap26)



www.openclassrooms.com

Dernière mise à jour le 5/02/2012

Sommaire

Créer un mini-RPG en JavaScript avec canvas

Le tutoriel que vous êtes en train de lire est en **bêta-test**. Son auteur souhaite que vous lui fassiez part de vos commentaires pour l'aider à l'améliorer avant sa publication officielle. Notez que le contenu n'a pas été validé par l'équipe éditoriale du Site du Zéro.

Par  Sébastien CAPARROS (sebcap26)

Mise à jour : 05/02/2012
Difficulté : Intermédiaire

Je vois régulièrement sur des forums d'informatique des novices qui souhaitent créer leurs propres jeux, et plus précisément leurs propres MMORPG (jeu de rôle massivement multijoueurs) en général.
C'est une tâche ardue, et j'ai moi même commencé par là. Je n'aborderai ici que l'aspect technique (programmation) du jeu.

Alors tu vas nous apprendre à créer un MMORPG complet ?

Non. Ce tutoriel a pour but de vous montrer les bases de la programmation d'un RPG en Javascript. Je n'aborderai pas l'aspect MMO, c'est à dire que ce sera un jeu en solo et pas multijoueurs. D'autre part, je rappelle que c'est un mini-RPG. Ce sera un jeu de rôle très basique et ne comportera globalement que la partie graphique du jeu.
Une fois les bases acquises, je pars du principe que vous saurez vous débrouiller seul. Il est donc important que vous alliez de l'avant en apportant vos propres modifications et essais au code que je propose. Le copié-collé sans chercher à comprendre ne vous servira à rien, et vous seriez probablement bloqué dès la fin du tuto.

Il n'y a pas qu'une seule méthode pour faire un jeu. Celle que je présente n'est peut être pas la meilleure, mais en tout cas c'est une méthode qui fonctionne, et c'est celle que j'utilise.

En résumé, voici ce que vous apprendrez à faire dans ce cours :

- Organiser les différentes données du jeu pour avoir un système aussi dynamique que possible
- Afficher une carte du monde
- Afficher des personnages qui peuvent se déplacer, et gérer l'animation de leurs déplacements
- Gérer la caméra
- Afficher des éléments d'interface (indicateur de vie par exemple)

Pour suivre ce cours, vous devez d'abord maîtriser :

- HTML
- JavaScript et sa programmation objet
- AJAX

Partie 1 : La carte

Le premier élément du jeu auquel nous allons nous intéresser est l'affichage de la carte.

Cette partie est sans doute la plus simple de notre RPG, mais c'est aussi la plus importante car elle pose les bases de la structure du jeu et vous permettra de vous familiariser avec l'élément canvas.

Mise en place des bases

Dans cette partie, nous allons poser les fondations de notre jeu.

Ce n'est pas une partie difficile, mais c'est la plus importante.

Structure des fichiers

Explications

Avant de commencer tout codage, je pense qu'il est important de mettre en place l'arborescence de notre application. Vous n'êtes bien sûr pas forcé d'utiliser la même que moi, mais au moins nous saurons avec précision où trouver ce que l'on cherche.

Ce tuto ne comportera aucun code exécuté côté serveur. Vous pouvez donc vous passer d'un serveur si vous n'en avez pas à disposition, mais il faudra peut-être configurer votre navigateur pour certaines parties (notamment pour autoriser l'utilisation de l'AJAX en mode hors ligne).

Notre application terminée ne comportera qu'un seul et unique fichier HTML. Je le nomme index.html pour des raisons pratiques, mais si vous intégrez plus tard notre mini-RPG dans un site plus complet, sachez que le nom de ce fichier n'a aucune importance.

Arborescence

J'ai également à la racine de mon application les dossiers suivants :

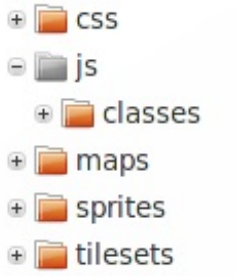
- css
- js
- maps
- sprites
- tilesets

Comme vous devez vous en douter, les dossiers js et css contiendront nos fichiers JavaScript et CSS. Le dossier maps comportera les fichiers définissant les cartes de notre jeu (j'ai l'habitude de les appeler "map", ce qui veut dire la même chose en Anglais. Si vous êtes amateur de jeux, ce terme devrait vous être familier.). Quant aux sprites et aux tilesets, j'expliquerai ça plus tard.
Dans mon dossier js, j'ai un sous-répertoire "classes". J'organiserai mon code comme ceci :

- Les fichiers directement placés dans js seront des fonctions et des bouts de codes divers
- Chaque classe JavaScript sera dans le dossier js/classes, et le nom du fichier sera celui de la classe, suivi de l'extension (.js)

Résultat

Vous devriez avoir quelque chose comme ceci :



Mise en place du canvas
Code de base

Bien, maintenant que nous avons mis en place notre arborescence, nous pouvons commencer la partie la plus intéressante : le codage. Ouvrez le fichier index.html (et créez le si ce n'est pas déjà fait) dans votre éditeur favori.

Copiez/collez la page suivante (elle est vide mais ça vous évitera de devoir tout taper) :

Code : HTML

```
<!DOCTYPE html>
<html>
 <head>
  <title>Mini-RPG</title>
 </head>
 <body>

 </body>
</html>
```

Vous avez peut-être remarqué le doctype un peu inhabituel. Ce n'est pas une erreur (et ce n'est pas non plus parce que j'ai la flemme). C'est le doctype qui a été adopté pour le HTML5 (il faut avouer que c'est quand même plus simple à mémoriser).

Nous allons maintenant y placer notre canvas.

Canvas

C'est quoi un canvas ?

Le canvas est une des nombreuses nouvelles possibilités offertes par HTML5. Concrètement ça fait pas mal de temps qu'elle existe déjà (c'est Apple qui l'a inventée, au sein de son navigateur Safari). Le canvas offre la possibilité de dessiner (carrés, rectangles, droites, texte, ronds, images ...) via du code JavaScript.

Ajoutez donc la ligne suivante dans le body de votre page HTML :

Code : HTML

```
<canvas id="canvas">Votre navigateur ne supporte pas HTML5, veuillez
le mettre à jour pour jouer.</canvas>
```

La phrase "Votre navigateur ne supporte pas HTML5, veuillez le mettre à jour pour jouer" placée dans la balise n'apparaîtra que sur les navigateurs concernés. En effet : si vous mettez quoi que ce soit dans la balise canvas, ce sera ignoré par les navigateurs compatibles. En revanche, les navigateurs qui ne connaissent pas cette balise vont tout simplement l'ignorer... et donc afficher

son contenu.

Si votre navigateur ne supporte pas canvas, une simple mise à jour devrait régler le problème. En revanche, si vous possédez Internet Explorer, cela ne suffira pas (sauf à partir de la version 9). Nous allons étudier son cas dans la partie qui suit.

Vous pouvez maintenant voir... Une page blanche. C'est pas beau la technologie ? C'est tout à fait normal étant donné que nous ne dessinons rien dans notre canvas pour le moment. Cependant, pour des raisons pratiques nous allons lui mettre un cadre.

Du CSS pour une meilleure visibilité

Créez donc un fichier style.css dans le dossier css. Ensuite liez-le à votre page HTML en mettant la ligne suivante dans votre head (rien de complexe si vous maîtrisez HTML) :

Code : HTML

```
<link rel="stylesheet" type="text/css" href="css/style.css" />
```

Pour afficher des bordures au canvas, il suffit de faire comme ceci :

Code : CSS

```
canvas {
 border: 1px solid black;
}
```

Le cas d'Internet Explorer
Quelle(s) solution(s) au problème ?

Si vous n'utilisez pas Internet Explorer, ce chapitre ne vous concerne pas personnellement. Cependant, il faut penser à vos futurs joueurs qui seront adeptes de ce navigateur.

Mais comment allons-nous faire s'il ne connaît pas canvas ?

Pour cela, nous allons devoir utiliser une librairie externe qui se nomme ExplorerCanvas.

Mise en place

Téléchargez la dernière version disponible sur le lien ci-dessus (à l'heure où j'écris ces lignes, le fichier se nomme excanvas_r3.zip) et dé-zippez le. Le fichier qui nous intéresse est excanvas.compiled.js.

À quoi sert le fichier excanvas.js ?

Le fichier excanvas.js est le code source de la librairie. L'autre aussi me direz vous, mais si vous ouvrez la version "compiled", vous verrez que le code n'est pas très digeste.

Les créateurs d'ExplorerCanvas ont tout simplement utilisé un obfuscateur de code. C'est un outil qui permet d'alléger le code source en supprimant les caractères inutiles au navigateur (commentaires, indentations, sauts de lignes), et de renommer les variables plus simplement (si vous survolez le code, vous verrez qu'il y a plusieurs variables et fonctions qui se nomment a, b, c...). Ainsi le fichier est plus rapide à charger et les curieux auront beaucoup de mal à copier le code source (ce qui n'est pas très utile pour cette librairie vu qu'elle est open source).

Mettez le fichier excanvas.compiled.js dans le dossier js de notre jeu. Ensuite, ajoutez cette ligne dans le head de la page html :

Code : HTML

```
<!--[if lt IE 9]><script type="text/javascript"
src="js/excanvas.compiled.js"></script><![endif]-->
```

Et voilà, maintenant ça fonctionne dans tous les navigateurs !

Explications

Mais comment les créateurs de cette librairie ont-ils fait pour dessiner des ronds, des carrés et des lignes ?

Internet Explorer implémente la technologie VML (Vector Markup Language) qui permet faire des choses semblables à canvas. Je ne sais pas vous donner plus de détails, mais je sais que les créateurs d'ExCanvas l'ont utilisée.

Sachez qu'il existe également d'autres librairies qui font le même travail qu'ExCanvas, mais c'est la seule que je connaisse qui ne fasse pas appel à Flash.

Utilisation de l'objet context

Quelques bases

Dans cette partie, je vais vous présenter un peu les possibilités de canvas. Nous n'attaquerons pas le jeu lui-même, c'est plutôt pour vous apprendre à vous familiariser avec l'objet "contexte".

Créez un fichier rpg.js (si vous trouvez un nom plus original n'hésitez pas ) à la racine du dossier js, et liez-le à votre page html (dans le head).

Ensuite placez-y le code suivant :

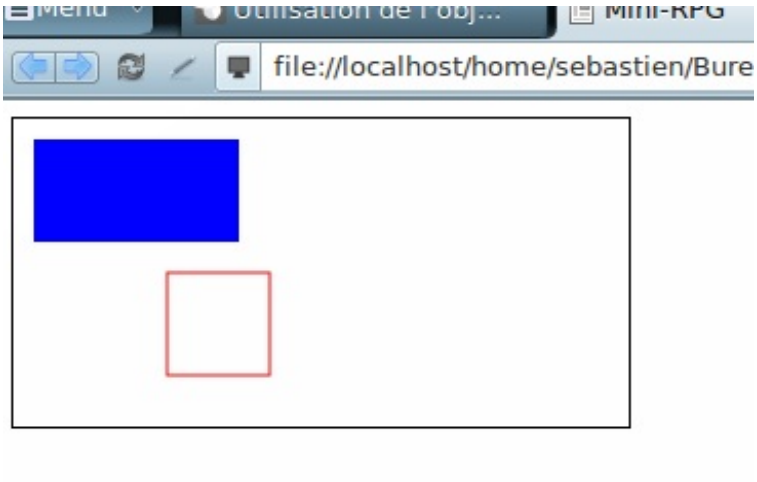
Code : JavaScript

```
window.onload = function() {
 var canvas = document.getElementById('canvas');
 var ctx = canvas.getContext('2d');

 ctx.fillStyle = 'blue';
 ctx.fillRect(10, 10, 100, 50);

 ctx.strokeStyle = 'red';
 ctx.strokeRect(75, 75, 50, 50);
}
```

Si tout va bien vous devriez obtenir ceci :



Explications sur le code

La première ligne n'a rien de difficile, elle nous permet de faire en sorte que notre code JavaScript ne soit exécuté qu'une fois la page chargée complètement. Ensuite nous récupérons l'élément DOM de notre canvas. Rien de nouveau non plus.

La ligne suivante est plus intéressante :

Code : JavaScript

```
var ctx = canvas.getContext('2d');
```

Elle nous permet de récupérer notre objet "context" (son nom est CanvasRenderingContext2D pour être plus précis).

Pourquoi on passe '2d' en paramètres ?

Le canvas est prévu pour être capable de faire bien plus que de la 2D. Le canvas permet déjà (sous les navigateurs Firefox 4 et Google Chrome 9) d'utiliser la technologie WebGL. Le webGL est une implémentation d'OpenGL pour JavaScript. Lorsque tous les navigateurs seront compatibles, vous pourrez donc voir fleurir des jeux 3D en JavaScript sans souci ni problèmes de performances puisque WebGL utilise l'accélération 3D.

Si vous souhaitez en savoir plus, je vous conseille d'aller voir du côté de c3dl.

Avant de continuer, je dois quand même préciser que le système de coordonnées de canvas (et de la grande majorité des langages de programmation) ne fonctionne pas dans le même sens qu'en mathématiques. Les axes fonctionnent dans ce sens :

Maintenant étudions la partie la plus intéressante du code :

Code : JavaScript

```
ctx.fillStyle = 'blue';
ctx.fillRect(10, 10, 100, 50);

ctx.strokeStyle = 'red';
ctx.strokeRect(75, 75, 50, 50);
```

Dans l'objet context, nous avons beaucoup de méthodes et d'attributs qui portent un nom commençant par 'fill' ou 'stroke'. Les méthodes en fill remplissent la forme que vous dessinez avec la couleur précisée dans l'attribut fillStyle. Les méthodes en stroke dessinent uniquement les contours des formes avec la couleur définie via strokeStyle.

Je ne détaillerai pas toutes les méthodes du context dans ce cours. Vous pouvez avoir plus de détails sur les méthodes du Context2D sur la page des spécifications officielles de HTML5 (en Anglais, attention la page est très lourde).

Traitement des images

Ces méthodes de dessin sont bien sympathiques, mais elles nous seront peu utiles pour notre RPG. Le dessin d'images par contre est très utile. Les méthodes de dessin du context nous permettent :

- De simplement dessiner une image
- De dessiner une image avec une taille différente
- De dessiner une partie d'une image

Ajoutez donc le code suivant à votre page pour voir concrètement ce que ça donne :

Code : JavaScript

```
ctx.drawImage(smiley, 200, 10);

ctx.drawImage(smiley, 200, 30, 100, 50);
ctx.drawImage(smiley, 0, 0, 10, 19, 200, 100, 10, 19);
```

Ainsi que celui-ci tout en haut du fichier (en dehors de tout bloc) :

Code : JavaScript

```
var smiley = new Image();
smiley.src =
"http://www.siteduzero.com/Templates/images/smilies/heureux.png";
```

Nous obtenons ceci :

En fonction de votre navigateur, il se peut que le smiley ne s'affiche pas. Si c'est le cas, téléchargez et enregistrez l'image du smiley dans votre application, et mettez un chemin relatif dans le code.

Dans la documentation, on peut voir qu'il y a surcharge de la méthode drawImage, ce qui nous donne trois possibilités :

Code : JavaScript

```
drawImage(image, dx, dy)
drawImage(image, dx, dy, dw, dh)
drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)
```

Plus de détails sur les noms des paramètres :

- d : Destination
- s : Source
- x : Coordonnée X
- Coordonnée Y
- w : Largeur
- h : Hauteur

La première se contente de dessiner l'image sans la modifier à partir des coordonnées du point en haut à gauche passé en paramètre. Dans la seconde, l'image est dessinée en fonction de la taille (dw et dh) passée en paramètre. Et dans la troisième, on précise une partie de l'image source que l'on souhaite (en précisant les coordonnées x et y, ainsi que la largeur et la hauteur du "rectangle" que l'on sélectionne. On précise la même chose pour la destination.

Vous pouvez préciser des hauteurs et largeurs de destination négatives, ce qui aura pour effet d'appliquer un "effet miroir" à votre image (elle sera inversée).

Dans cette partie, nous n'avons pas encore réellement commencé le RPG.

Néanmoins, poser les bases était nécessaire. Le canvas est primordial pour notre RPG. N'hésitez donc pas à vous entraîner pour bien le maîtriser, ainsi que les méthodes du context avant de poursuivre ce cours.

Gestion des tilesets

Dans cette partie, nous allons voir ce qu'est un tileset, à quoi ca sert et comment l'utiliser dans notre RPG.

Présentation

Un tileset, c'est quoi ?

Dans un RPG, une carte (map) est constituée de cases que l'on nomme généralement tiles (tuiles en français). C'est en plaçant plusieurs tiles les uns à côté des autres que l'on va générer notre monde.

Chaque tile est une image carrée. On utilise généralement une taille de 32 x 32 pixels.
Dans ce tuto, c'est également la taille que j'utiliserai car vous pourrez trouver une énorme quantité de ressources graphiques libres sur internet basées sur ce modèle (pour ceux qui connaissent RPG Maker, sachez que c'est le même format).

Si vous avez une carte très basique avec une étendue d'herbe et un chemin qui la traverse, vous aurez globalement deux tiles différents : un avec une texture d'herbe et un avec une texture de chemin. Dans la pratique c'est un peu plus compliqué. On utilisera généralement une dizaine de tiles afin de gérer les bordures du chemin (sinon il sera totalement carré, ce qui n'est pas très beau).

Pourquoi un tel système ?

La première chose qui nous vient à l'esprit quand on a pas l'expérience du développement de jeux en 2D est de tout simplement avoir un fichier image de 32 x 32 pixels par tile.

Cette méthode présente un inconvénient non négligeable en terme de performances. Imaginez que vous ayez une très grande carte sur laquelle vous avez une centaine de tiles différents. Cela veut dire que vous allez devoir charger une centaine d'images, et donc - si votre jeu est sur un serveur web - une centaine de requêtes HTTP.

L'objectif du tileset est de combiner plusieurs tiles sur une seule image. Ainsi, même si cela n'a pas d'impact en terme de taille, ça en aura en performances car il n'y aura qu'une seule image à charger.

Pour cette partie, nous allons utiliser un tileset très simple qui est celui-ci :

Ce tileset est gratuit mais tout de même soumis à une licence. Aussi, je vous informe que je l'ai pris ici et qu'il a été créé par des personnes nommées "qubodup", "Bart K." et "Blarumyrran".

Vous comprenez mieux ce qu'est un tileset à présent ?

Mais comment allons-nous l'utiliser dans notre jeu puisqu'il n'est pas découpé ?

Vous vous souvenez de la méthode drawImage de l'objet context vu dans la partie précédente ? Nous avons vu qu'elle pouvait nous permettre de ne dessiner qu'une partie de l'image en sélectionnant un "rectangle" source par ses coordonnées (x, y) et par sa taille.

C'est ce que nous allons maintenant étudier.

Découpage et utilisation

Structure de la classe

Avant de commencer le codage du côté des tilesets, nous allons-nous mettre d'accord sur la manière de les appeler. Comment allons nous choisir de dessiner tel ou tel tile ? Plusieurs solutions se présentent à nous. La première qui nous vient à l'esprit (et

qui est la plus facile à calculer) est de les appeler par des coordonnées (x, y), x et y étant non pas des pixels mais des tiles.

Ce n'est pas la convention de numérotation que j'ai choisie. J'ai choisi de n'utiliser qu'un seul entier. Ce sera moins simple à calculer, mais ça nous simplifiera la vie plus tard. Voici la manière dont ce sera numéroté pour notre tileset :

Bien. Maintenant commençons un peu à coder.

Premièrement, enregistrez l'image de tileset dans le répertoire "tilesets" que nous avons créé dans la première partie. Je l'ai nommée "basique.png". Si vous choisissez un nom différent, pensez bien à reporter vos modifications dans le code.

Programme de test

Reprenez le fichier rpg.js et faites un peu de ménage (si vous avez fait des tests que vous souhaitez garder, n'oubliez pas de les sauvegarder dans un coin) :

Code : JavaScript

```
window.onload = function() {
 var canvas = document.getElementById('canvas');
 var ctx = canvas.getContext('2d');

 // Nous allons insérer nos tests ici
}
```

Pour gérer nos tilesets, nous allons utiliser une classe. Nous aurons besoin des méthodes suivantes :

- Constructeur(String)
- dessinerTile(int, Context, int, int)

Le constructeur prendra en paramètres le nom de l'image contenant le tileset.
La méthode dessinerTile nous permettra de facilement dessiner un tile (dont le numéro est passé en premier paramètre) sur un context passé en second paramètre, aux coordonnées x et y (troisième et quatrième paramètre).

Maintenant que nous avons défini notre classe, nous allons mettre en place nos tests, comme ça ce sera fait. Remplacez le commentaire que j'ai laissé à cet effet dans rpg.js par le code suivant :

Code : JavaScript

```
ts.dessinerTile(1, ctx, 10, 10);
ts.dessinerTile(5, ctx, 50, 10);
ts.dessinerTile(6, ctx, 90, 10);
ts.dessinerTile(7, ctx, 130, 10);
```

Et ajoutez la ligne suivante tout en haut (en dehors de la fonction) :

Code : JavaScript

```
var ts = new Tileset("basique.png");
```

Nous créons une instance de tileset en dehors de la fonction window.onload. Ainsi, le navigateur commencera à charger l'image en même temps que le reste de la page. La fonction définie via window.onload ne sera donc appelée qu'une fois l'image totalement chargée (et donc prête à être dessinée).

Vous l'aurez sans doute compris, notre test va dessiner côte à côte les tiles numéros 1, 5, 6 et 7. Ce jeu d'essai teste principalement les "coins" du tileset, car c'est souvent les coins qui posent problème.

Création de notre classe

Nous allons maintenant passer au codage de notre classe. Créez donc un fichier Tileset.js dans le dossier js/classes, et liez-le à votre fichier HTML :

Code : HTML

```
<script type="text/javascript" src="js/classes/Tileset.js"></script>
```

Nous pouvons dès lors poser les bases de notre classe sans trop de difficulté :

Code : JavaScript

```
function Tileset(url) {
 // Chargement de l'image dans l'attribut image
 this.image = new Image();
 this.image.referenceDuTileset = this;
 this.image.onload = function() {
  if(!this.complete)
   throw new Error("Erreur de chargement du tileset nommé \"" + url
+ "\".");
 }
 this.image.src = "tilesets/" + url;
}

// Méthode de dessin du tile numéro "numero" dans le contexte 2D
"context" aux coordonnées x et y
Tileset.prototype.dessinerTile = function(numero, context,
xDestination, yDestination) {

}
```

Dans le constructeur, vous voyez qu'on ajoute un attribut spécial à notre objet image afin qu'il sache à quel objet Tileset il appartient. Ainsi, nous pourrons terminer le constructeur de notre Tileset de manière asynchrone, c'est à dire quand l'image sera chargée.

Dessin d'un tile

Ici, le constructeur n'est pas terminé. Pour pouvoir déterminer les coordonnées (x, y) d'un tile précis dans le tileset à partir de son numéro, nous allons avoir besoin de connaître la largeur en tiles de notre tileset. Il faut donc ajouter le code suivant dans la fonction onload de l'image :

Code : JavaScript

```
// Largeur du tileset en tiles
this.referenceDuTileset.largeur = this.width / 32;
```

Nous allons pouvoir maintenant attaquer notre méthode de dessin.
Voici un tableau récapitulatif des paramètres dont nous avons besoin pour utiliser la méthode drawImage du context, ainsi que leur description et leur valeur si elle est déjà connue :

Paramètre	Description	Valeur
image	L'image source	this.image
sx	Coordonnée x du tile dans le tileset	???
sy	Coordonnée y du tile dans le tileset	???
sw	Largeur de l'image source	32
sh	Hauteur de l'image source	32
dx	Coordonnée x de destination	xDestination
dy	Coordonnée y de destination	yDestination
dw	Largeur de l'image a dessiner	32
dh	Hauteur de l'image a dessiner	32

Nous avons donc à peu près ce qu'il nous faut, sauf les coordonnées x et y du tile demandé dans le tileset. Nous allons donc déterminer sa position (x, y), **en nombre de tiles**.

Tout d'abord, la coordonnée x :

Code : JavaScript

```
var xSourceEnTiles = numero % this.largeur;
if(xSourceEnTiles == 0) xSourceEnTiles = this.largeur;
```

Ici, il n'y a que des maths. On prend le reste de la division entière du numéro du tile par la largeur (en tiles) du tileset. On obtient alors le numéro de la colonne où est situé le tile. Par contre si on obtient 0, c'est qu'on est sur un tile tout à droite du tileset.

Pour la coordonnée y :

Code : JavaScript

```
var ySourceEnTiles = Math.ceil(numero / this.largeur);
```

Ici on divise le numéro demandé par la largeur en tiles. Si nous obtenons un nombre entier, pas de problème. Dans le cas contraire c'est que nous ne sommes plus sur la ligne indiquée par la partie entière, mais sur la ligne suivante.

Maintenant, c'est plutôt facile :

Code : JavaScript

```
var xSource = (xSourceEnTiles - 1) * 32;
var ySource = (ySourceEnTiles - 1) * 32;
```

Il faut bien penser à ne pas oublier le -1. En effet, le tile numéro 1 est situé aux coordonnées (0, 0), pas (32, 32).

Il ne nous reste plus qu'à utiliser la méthode de dessin :

Code : JavaScript

```
context.drawImage(this.image, xSource, ySource, 32, 32,
xDestination, yDestination, 32, 32);
```

Si tout va bien vous devriez obtenir ceci, qui correspond bien a nos attentes :

Nous sommes maintenant capables d'afficher facilement n'importe quel tile de notre tileset !

Vous verrez que ca sera bien plus facile pour afficher notre carte dans la partie suivante.

Mise en place du terrain

Nous allons maintenant mettre en place la première partie de la carte : le terrain.

Cette partie vous permettra de vous rendre compte de l'utilité des tilesets que nous avons mis en place dans la partie précédente, ainsi que de la facilité que ça nous apporte.

Format de stockage des données

Choix d'un langage de représentation des données

Nous allons tout d'abord commencer par nous interroger sur la manière dont nous allons stocker nos cartes, et plus précisément sur la syntaxe que nous allons utiliser.

À première vue, il existe plusieurs solutions que je vais répertorier ici :

Format	Avantages	Inconvénients
XML	• Possibilité de structurer correctement notre document • Facile à interpréter • Standard	• Les données seront difficilement lisibles dans notre cas (c'est à dire une liste de cases à deux dimensions). • Syntaxe très lourde, le fichier sera d'autant plus gros
JSON	• Syntaxe légère • Standard • Très lisible • Facile à interpréter	
Texte brut	• Syntaxe légère • Très lisible	• Il faudra développer notre propre interpréteur • Moins facile et moins propre d'intégrer des valeurs supplémentaires
Binaire	• Très très léger	• Complètement illisible visuellement • Moins facile et moins propre d'intégrer des valeurs supplémentaires • JavaScript est prévu pour traiter des chaînes, pas du binaire. Outre les difficultés que cela impliquerait, nous y perdrions les gains en performances
CSV	• Léger • Standard • Éditable dans un tableur • Interpréteur facile à faire ou à trouver.	• Moins facile et moins propre d'intégrer des valeurs supplémentaires

Le format que j'ai choisi pour notre RPG est le JSON.

Structure de nos cartes

Dans notre fichier, nous allons devoir stocker pour le moment les éléments suivants (mais il y en aura d'autres à l'avenir) :

- Terrain présent sur chaque case
- Tileset qu'on va utiliser
- Taille de la carte

Pour la taille de la carte, ça va être facile : nos cases seront stockées dans des tableaux, il suffira donc de récupérer la taille des tableaux.

Avant de poursuivre, je vous invite à enregistrer le tileset que nous utiliserons ici (je l'ai pris sur rpg-maker.fr, puis modifié un peu pour l'adapter à nos besoins), qui est le suivant (je l'ai nommé "chemin.png") :

Voici le code de notre première carte (enregistrez-le dans le dossier "maps", avec l'extension ".json". Pour ma part, il se nomme "premiere.json".) :

Code : JavaScript

```
{
 "tileset" : "chemin.png",
 "terrain" : [
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10,  8,  6,  6,  6,  6,  6,  6],
  [2, 2, 2, 2, 2, 2, 9, 10, 10, 10, 10, 10, 10, 10, 10],
  [2, 2, 2, 2, 2, 2, 9, 10, 12, 14, 14, 14, 14, 14, 14],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2],
  [2, 2, 2, 2, 2, 2, 9, 10, 11,  2,  2,  2,  2,  2,  2]
 ]
}
```

Pour ceux qui sont courageux ~~ou masochistes (comme moi)~~, vous verrez peut-être dès maintenant à quoi ressemblera la carte. Vous pouvez même la modifier. Comme vous pouvez le voir, cette syntaxe nous permet presque d'avoir un aperçu en deux dimensions du code de la carte, ce qui plus facile à manipuler.

Et tu sors ce code d'où ?

J'ai pris le temps de le faire en mode texte (et c'était d'ailleurs un peu pénible ). Lorsque l'ensemble du tutoriel sera fini, avec tout ce que vous aurez appris, vous devriez être capable de facilement créer un éditeur de cartes pour tout faire visuellement.

Chargement et affichage du terrain

Structure de la classe

Nous pouvons maintenant passer au codage à proprement parler.

Nous allons créer une classe "Map" (mais vous pouvez la nommer "Carte" si vous préférez l'avoir en français, néanmoins le terme de "map" vous sera plus familier si vous avez un peu l'habitude des jeux vidéo). Cette classe aura pour rôle de charger les données de la carte (que nous avons mises en place dans la partie précédente), et d'afficher le terrain sur demande.

Un peu comme pour le tileset, nous aurons donc une méthode "dessinerMap", qui prendra en paramètre un objet context. Cette méthode y dessinera l'ensemble du terrain.
J'y ajouterai également deux getters : getHauteur() et getLargeur().

Programme de test

Ouvrez votre fichier rpg.js et mettez-y le code suivant (et pensez à faire une sauvegarde du code précédent s'il y a quelque chose que vous souhaitez conserver) :

Code : JavaScript

```
var map = new Map("premiere");

window.onload = function() {
 var canvas = document.getElementById('canvas');
 var ctx = canvas.getContext('2d');

 canvas.width  = map.getLargeur() * 32;
 canvas.height = map.getHauteur() * 32;

 map.dessinerMap(ctx);
}
```

Comme vous pouvez le voir, nous modifions automatiquement la taille du canvas pour que l'ensemble de la carte puisse y tenir. Cette méthode n'est pas la meilleure puisqu'elle nous posera des soucis lorsqu'on voudra faire une carte plus grande que l'écran. Nous allons quand même l'utiliser pour le moment, mais nous y reviendrons dans une prochaine partie.

Chargement des données JSON

Maintenant créez notre classe Map dans le fichier qui va bien, et pensez à l'inclure depuis le fichier HTML.

Notre constructeur chargera les données de la carte, et interprétera le code JSON.
Nous allons donc utiliser AJAX (ou XMLHttpRequest pour les intimes). Pour nous simplifier la vie, je vais réutiliser la fonction que Thunderseb a mis a disposition dans son tuto sur Ajax. Enregistrez la dans le dossier js (et pensez bien à l'inclure dans votre page HTML).
Nous allons donc commencer par créer une instance de cet objet :

Code : JavaScript

```
function Map(nom) {

 // Création de l'objet XmlHttpRequest
 var xhr = getXMLHttpRequest();

 // Ici viendra le code que je vous présente ci-dessous
}
```

Ensuite, nous lui demandons de charger le contenu du fichier JSON :

Code : JavaScript

```
// Chargement du fichier
xhr.open("GET", './maps/' + nom + '.json', false);
xhr.send(null);
if(xhr.readyState != 4 || (xhr.status != 200 && xhr.status != 0)) //
Code == 0 en local
```

```
        throw new Error("Impossible de charger la carte nommée \"" + nom +
"\" (code HTTP : " + xhr.status + ").");
var mapJsonData = xhr.responseText;
```

Si vous n'utilisez pas de serveur web, il faudra peut-être configurer votre navigateur pour qu'il accepte d'utiliser AJAX sur des fichiers locaux (c'est notamment le cas d'Opera).

Vous noterez que l'objet ne renvoie pas un code HTTP 200 si vous êtes en local puisque vous n'utilisez pas HTTP dans ce cas là.

Nous avons maintenant le contenu de notre fichier (donc le code JSON) dans la variable mapJsonData. Il nous suffit de le "parser", c'est à dire de le transformer en objet(s) JavaScript. Pour cela il existe plusieurs méthodes. La plus propre, c'est d'utiliser la méthode JSON.parse. Mais sur certains navigateurs un peu anciens, cette méthode n'existe pas forcément. Pour eux, il va falloir inclure la librairie JavaScript qui convient (disponible sur le site http://www.json.org/, ou si vous préférez le lien direct vers le fichier, qu'il faudra inclure dans votre HTML, de préférence en premier) :

Code : JavaScript

```
// Analyse des données
var mapData = JSON.parse(mapJsonData);
```

JSON étant un langage de représentation tiré de JavaScript, c'est très performant car totalement natif.
Nous avons donc maintenant une variable mapData, qui fait référence à un objet contenant :

- Un attribut tileset, qui contient le nom du tileset à utiliser
- Un attribut terrain, qui contient une matrice (ou tableau a deux dimensions) contenant des entiers. Chaque entier est le numéro du tile a utiliser pour la case a laquelle il correspond.

Nous pouvons maintenant charger le tileset requis et stocker la matrice du terrain dans un attribut de notre objet :

Code : JavaScript

```
this.tileset = new Tileset(mapData.tileset);
this.terrain = mapData.terrain;
```

Mise en place des méthodes de l'objet

Viennent ensuite nos trois méthodes. Les deux getters sont les plus simples :

Code : JavaScript

```
// Pour récupérer la taille (en tiles) de la carte
Map.prototype.getHauteur = function() {
 return this.terrain.length;
}
Map.prototype.getLargeur = function() {
 return this.terrain[0].length;
}
```

La hauteur est définie par le nombre de tableaux (donc de lignes) présents dans le tableau principal.
La largeur est égale à la taille de n'importe quelle ligne (ici j'ai choisi la première puisqu'on a forcément au moins une ligne).

La méthode dessinerMap suivra l'algorithme suivant :

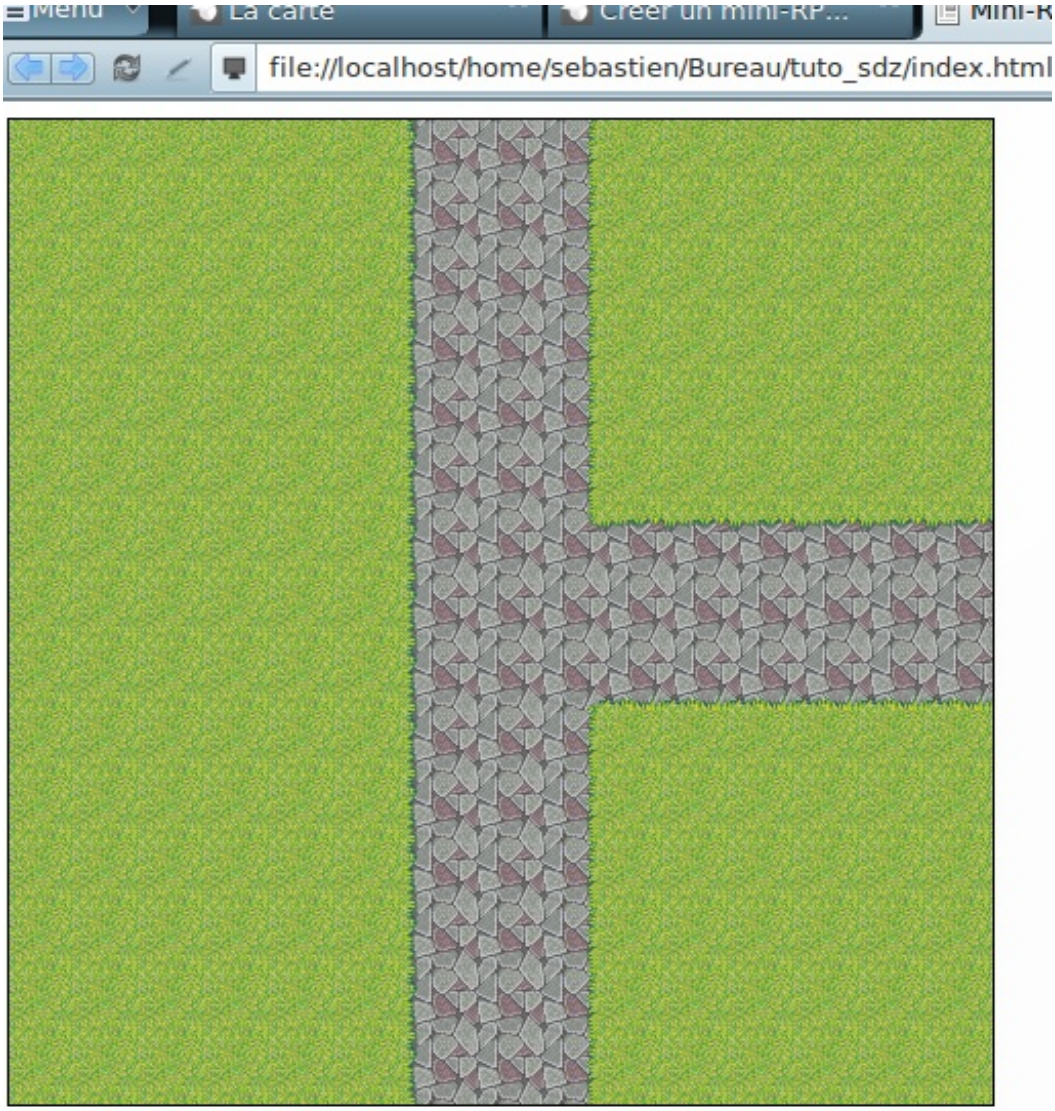
- On parcourt toutes les lignes
- Pour chaque ligne, on parcourt toutes les cellules

- Ensuite on demande au tileset de dessiner le tile dont le numéro est disponible dans la cellule, en lui passant les coordonnées correspondantes, c'est à dire (X = indice de la colonne * 32, Y = indice de la ligne * 32).

Code : JavaScript

```
Map.prototype.dessinerMap = function(context) {
 for(var i = 0, l = this.terrain.length ; i < l ; i++) {
  var ligne = this.terrain[i];
  var y = i * 32;
  for(var j = 0, k = ligne.length ; j < k ; j++) {
   this.tileset.dessinerTile(ligne[j], context, j * 32, y);
  }
 }
}
```

Si tout va bien, vous devriez maintenant voir ceci :



Nous sommes maintenant capables d'afficher aisément un terrain. Bien sûr c'est un peu vide, mais patience, nous y ajouterons des choses.

Et voilà, nous avons terminé la première partie de notre mini-RPG.

Dans la prochaine partie nous mettrons plus de choses sur notre terrain.

Vous pouvez télécharger le code source de cette partie ici, ou bien le tester ici.

Partie 2 : Les personnages et les décors

Dans cette partie, nous mettrons en place les personnages du jeu et leurs déplacements.

Nous ajouterons également des décors au jeu et étudierons les interactions et les événements entre tout ce petit monde.

Les personnages

Dans cette partie, nous allons commencer à intégrer un personnage (celui que le joueur dirige, mais une bonne partie du travail sera faite pour les PNJ (Personnages Non Joueurs)) au jeu. Nous étudierons comment faire en sorte de pouvoir le déplacer à l'écran.

À la fin de cette partie, le personnage ne se déplacera que très basiquement, c'est à dire qu'il se "téléportera" d'une case à l'autre. L'animation du déplacement se fera dans la prochaine partie.

Choix et rendu des sprites

Le fichier de sprite

Avant d'afficher nos personnages, il faut se mettre d'accord sur le format utilisé.

C'est quoi un sprite ?

Non non, il ne s'agit pas d'une boisson, mais d'une image. Dans la même idée que le tileset, le sprite permet de combiner toutes les images d'un personnage dans un seul fichier afin d'en réduire le nombre.

Comme ce n'est probablement pas clair, voici en image le format que j'ai utilisé pour ce tutoriel. Pour les connaisseurs, vous verrez qu'il s'agit du format utilisé par RPG maker (un format que je trouve assez facile à comprendre, mais surtout qui vous permettra de trouver beaucoup de ressources sur Internet) :

(image tirée du site rpg-maker.fr)

Nous voyons donc que cette image en contient en réalité 16 (4 lignes et 4 colonnes). Étudions-la de plus près :

- Sur la première ligne, il y a 4 images du personnage tourné vers le bas
- Sur la seconde ligne, il y a 4 images du personnage tourné vers la gauche
- Sur la troisième ligne, il y a 4 images du personnage tourné vers la droite
- Sur la dernière ligne, il y a 4 images du personnage tourné vers le haut

Ensuite, pour chaque ligne, nous avons quatre images. Ces quatre images constitueront les quatre "frames" de l'animation du personnage (que nous étudierons un peu plus loin). Ce qui nous importe pour le moment, c'est la colonne de gauche. C'est là que l'on trouve l'image du personnage lorsqu'il est immobile.

Notez qu'il peut exister des images de sprites de différentes tailles, petites ou grandes.

Avant de passer à la suite, je vous invite à enregistrer notre personnage dans le dossier "sprites" que nous avions créé au début (je l'ai nommé "exemple.png", mais si vous avez une idée de nom plus original n'hésitez pas).

La classe Personnage

Nos personnages auront quatre attributs :

- Un pour l'image du sprite
- Deux dans lesquels nous stockerons la taille de chaque partie de l'image (pour la calculer une seule fois dans le constructeur)
- Deux pour la position du personnage en x et y
- Un pour l'orientation du personnage

C'est ce dernier auquel nous allons nous intéresser. Il va falloir choisir quoi mettre dans notre variable. Plusieurs solutions sont alors possibles, comme stocker des chaînes de caractères ou des objets. Ce serait plus facile à gérer (quoique ...), mais ce serait tout sauf performant.
La solution pour laquelle j'ai opté, c'est d'utiliser des nombres entiers. C'est moins facile à lire pour nous, mais les performances seront meilleures. Seulement, il faut là aussi qu'on choisisse quel entier associer à quelle direction. Nous pourrions nous faciliter la vie en prenant des nombres logiques, allant par exemple dans le sens des aiguilles d'une montre (1 pour le haut, 2 pour la droite ...). Saut qu'à un moment donné, lorsque nous voudrons dessiner notre personnage, il faudra déterminer quelle portion de l'image prendre par rapport à cet attribut. Si nous prenions des entiers comme ceux-là, nous devrions avoir quatre conditions (if, else if, ou switch) pour trouver les coordonnées de cette portion.

Nous pouvons l'éviter en prenant des nombres ordonnés dans la même logique que le fichier de sprite. Nous aurons donc :

- 0 pour le personnage orienté vers le bas
- 1 pour le personnage orienté vers la gauche
- 2 pour le personnage orienté vers la droite
- 3 pour le personnage orienté vers le haut

Ainsi, nous pourrons déterminer la position de la ligne correspondant à la direction sur l'image en multipliant simplement la hauteur d'une ligne par l'entier représentant la direction du personnage.

Afin que ce soit plus facile à utiliser, je vous propose de définir des constantes (je les place dans le même fichier que la classe Personnage, mais tout en haut, en dehors de la classe elle-même).
Nous pouvons donc commencer à créer le fichier Personnage.js, dans le dossier js/classes (en pensant bien à l'inclure dans la page html) :

Code : JavaScript

```
var DIRECTION = {
 "BAS"    : 0,
 "GAUCHE" : 1,
 "DROITE" : 2,
 "HAUT"   : 3
}

// Ici se trouvera la classe Personnage
```

Étant donné que nous avons tout ce qu'il faut, nous pouvons commencer à mettre en place la classe Personnage elle-même (le constructeur n'ayant pas grand-chose de plus complexe que pour la classe Tileset, je ne vais pas expliquer ce code, notamment pour ce qui est du chargement de l'image) :

Code : JavaScript

```
function Personnage(url, x, y, direction) {
 this.x = x; // (en cases)
 this.y = y; // (en cases)
 this.direction = direction;
```

```
 // Chargement de l'image dans l'attribut image
 this.image = new Image();
 this.image.referenceDuPerso = this;
 this.image.onload = function() {
  if(!this.complete)
   throw "Erreur de chargement du sprite nommé \"" + url + "\".";

  // Taille du personnage
  this.referenceDuPerso.largeur = this.width / 4;
  this.referenceDuPerso.hauteur = this.height / 4;
 }
 this.image.src = "sprites/" + url;
}

Personnage.prototype.dessinerPersonnage = function(context) {
 // Ici se trouvera le code de dessin du personnage
}
```

Comme vous le voyez, la méthode dessinerPersonnage n'a besoin que d'un seul paramètre, qui est l'objet context sur lequel il faut dessiner le personnage, car nous avons toutes les données nécessaires pour dessiner notre personnage :

Code : JavaScript

```
context.drawImage(
 this.image,
 0, this.direction * this.hauteur, // Point d'origine du rectangle
source à prendre dans notre image
 this.largeur, this.hauteur, // Taille du rectangle source (c'est
la taille du personnage)
 (this.x * 32) - (this.largeur / 2) + 16, (this.y * 32) -
this.hauteur + 24, // Point de destination (dépend de la taille du
personnage)
 this.largeur, this.hauteur // Taille du rectangle destination
(c'est la taille du personnage)
);
```

Il y a deux calculs importants dans ce code.

- Le point x où nous allons dessiner notre personnage. Il faut y soustraire la moitié de la largeur du personnage. Autrement, si le personnage est plus large que la taille d'une case, il apparaîtra à droite de cette case. Il faut également y ajouter 16 (c'est-à-dire la moitié de la largeur d'une case), sinon il sera centré, mais par rapport au côté gauche de la case.
- Le point y où nous allons dessiner notre personnage. Il faut y soustraire la hauteur du personnage et y ajouter 24 (les trois quarts de la hauteur d'une case, car si vous observez bien le sprite, il y a toujours une petite marge qu'il faut prendre en compte entre les pieds du personnage et le bas de l'image, mais sinon on aurait pris 32). Autrement, comme la plupart de nos personnages sont plus hauts qu'une case, c'est leur tête qui apparaîtrait au niveau de la case, alors que ce sont leurs pieds qui touchent le sol.

Intégration basique des personnages dans la carte

Il ne nous manque plus qu'un détail pour afficher nos personnages : décider où faire appel à cette méthode de dessin. J'ai choisi d'intégrer dans la classe Map une liste des personnages qu'il faut afficher. Comme plus tard nous aurons des décors, et que ces décors pourront aussi bien être au dessus qu'en dessous des personnages, il est plus simple de le faire comme ça. De plus, comme nous aurons des PNJ (personnages non joueurs : vendeurs, passants ...) qui seront décrits dans le code JSON de chaque map, il est plus simple que ce soit la map qui gère cela.

Pour le moment, nous allons donc utiliser un simple tableau en attribut pour cela (j'ai mis ce code juste au-dessus des déclarations de méthodes, mais vous pouvez le mettre où vous voulez dans le constructeur) :

Code : JavaScript

```
// Liste des personnages présents sur le terrain.
this.personnages = new Array();
```

Il nous faut aussi une méthode pour remplir ce tableau :

Code : JavaScript

```
// Pour ajouter un personnage
Map.prototype.addPersonnage = function(perso) {
 this.personnages.push(perso);
}
```

Et enfin, il faut que la méthode de dessin de la map dessine également nos personnages. Pour cela, j'ajoute (temporairement) le code suivant à la suite de la boucle, en bas de la méthode dessinerMap :

Code : JavaScript

```
// Dessin des personnages
for(var i = 0, l = this.personnages.length ; i < l ; i++) {
 this.personnages[i].dessinerPersonnage(context);
}
```

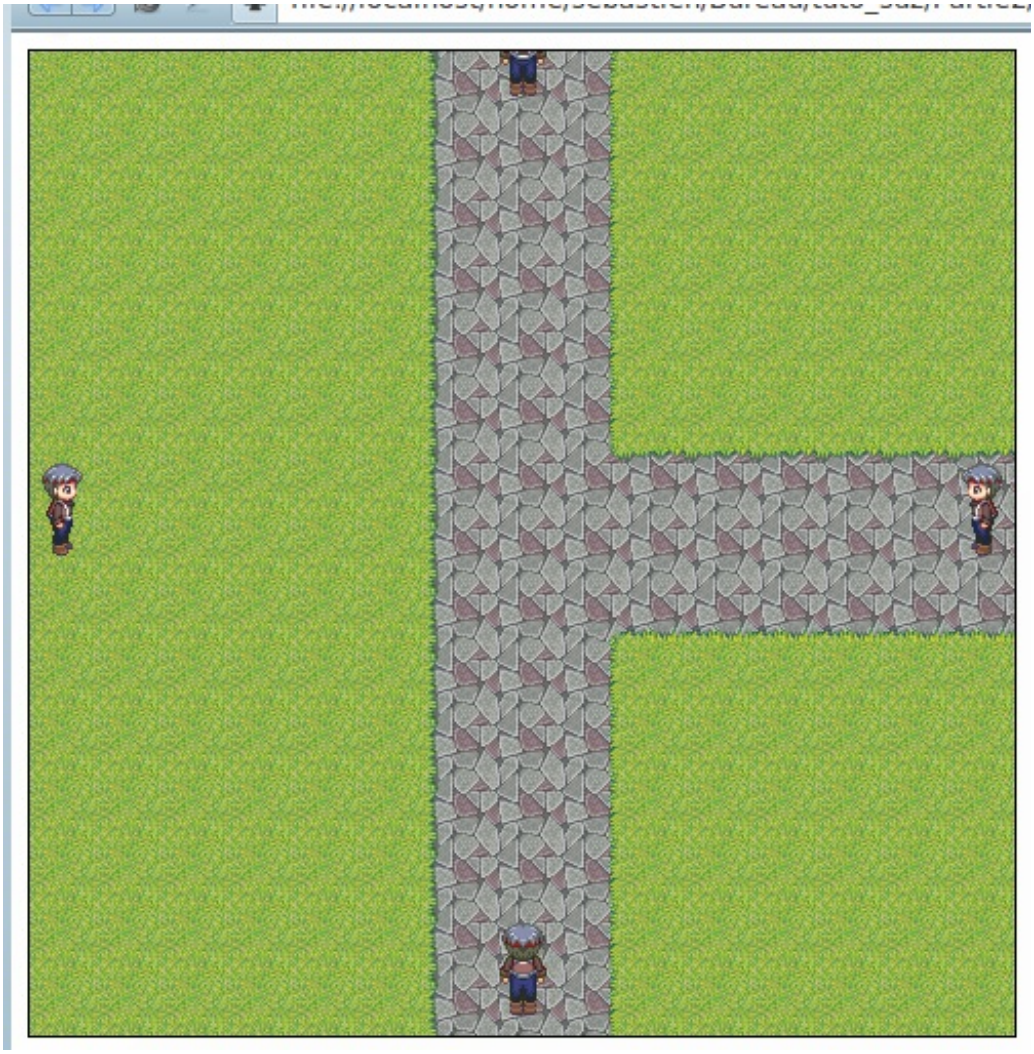
Ce code n'est pas du tout optimisé. C'est pour cela qu'il n'est que temporaire. Quand nous passerons aux décors, nous optimiserons cela en structurant mieux la liste des personnages. Pour le moment je ne peux pas expliquer pourquoi sans dévoiler la partie sur la gestion des décors.

Nous pouvons maintenant tester le dessin de nos personnages. Pour cela, je vais créer quatre personnages, chacun orienté dans une direction précise. Et pour être sûr qu'ils se positionnent aux bons endroits, je vais les placer sur les côtés de la carte, le regard tourné vers le centre. Pour cela, j'ajoute le code suivant au fichier rpg.js, tout en haut du fichier (mais en dessous de la ligne qui instancie la carte) :

Code : JavaScript

```
map.addPersonnage(new Personnage("exemple.png", 7, 14,
DIRECTION.HAUT));
map.addPersonnage(new Personnage("exemple.png", 7, 0,
DIRECTION.BAS));
map.addPersonnage(new Personnage("exemple.png", 0, 7,
DIRECTION.DROITE));
map.addPersonnage(new Personnage("exemple.png", 14, 7,
DIRECTION.GAUCHE));
```

Si tout va bien vous devriez obtenir ceci :



Clavier et boucle principale
Méthode de déplacement

Maintenant que nous pouvons afficher nos personnages, il fa falloir qu'ils puissent se déplacer. Pour le moment nous allons faire des déplacements très simples, c'est-à-dire que le personnage passera d'une case à l'autre sans animation.

Pour cela, nous allons avoir une seule méthode, à laquelle il faudra passer deux paramètres :

- La direction vers laquelle on souhaite aller
- La référence de l'objet map. Nous en aurons besoin pour faire plusieurs vérifications. Il faudra en premier lieu savoir si le joueur n'essaye pas d'aller en dehors de la carte, mais plus tard il faudra vérifier qu'il peut aller sur la case en question (est-ce un mur par exemple ?)

Dans tous les cas (que le personnage puisse aller sur la case désirée ou non), il faudra changer la direction du personnage. Entre autres, si le joueur ne comprend pas pourquoi le déplacement ne se fait pas, il saura au moins que l'appui sur la touche a bien été détecté, car le personnage ne regardera plus au même endroit.

Nous allons avoir besoin de connaître la case où l'on souhaite aller, à partir des coordonnées courantes et de la direction souhaitée. Je vais pour cela créer une méthode, qui prend en paramètre une direction, et qui renvoie un objet contenant deux attributs (x et y), qui sont les coordonnées correspondant à la case située dans la direction donnée par rapport au point actuel du joueur. Cette méthode nous sera utile à plusieurs reprises (et ce sera plus propre car nous aurons obligatoirement besoin de faire plusieurs tests).

Voici donc nos deux méthodes :

Code : JavaScript

```
Personnage.prototype.getCoordonneesAdjacentes = function(direction)
{
 var coord = {'x' : this.x, 'y' : this.y};
 switch(direction) {
  case DIRECTION.BAS :
   coord.y++;
   break;
  case DIRECTION.GAUCHE :
   coord.x--;
   break;
  case DIRECTION.DROITE :
   coord.x++;
   break;
  case DIRECTION.HAUT :
   coord.y--;
   break;
 }
 return coord;
}

Personnage.prototype.deplacer = function(direction, map) {
 // On change la direction du personnage
 this.direction = direction;

 // On vérifie que la case demandée est bien située dans la carte
 var prochaineCase = this.getCoordonneesAdjacentes(direction);
 if(prochaineCase.x < 0 || prochaineCase.y < 0 || prochaineCase.x >=
map.getLargeur() || prochaineCase.y >= map.getHauteur()) {
  // On retourne un booléen indiquant que le déplacement ne s'est
pas fait,
  // Ça ne coute pas cher et ca peut toujours servir
  return false;
 }

 // On effectue le déplacement
 this.x = prochaineCase.x;
 this.y = prochaineCase.y;

 return true;
}
```

Le clavier

Nous allons maintenant modifier le fichier rpg.js. Nous allons supprimer nos quatre personnages de test précédents (les quatre appels à la méthode "addPersonnage), pour n'en mettre qu'un seul, qui sera le personnage du joueur :

Code : JavaScript

```
var joueur = new Personnage("exemple.png", 7, 14, DIRECTION.BAS);
map.addPersonnage(joueur);
```

Nous allons maintenant nous intéresser à la gestion du clavier. Je ne vais pas faire un système "dynamique", c'est-à-dire que l'association d'une touche avec la fonction qu'elle déclenche sera écrite directement dans le code (mais rien ne vous empêche de faire quelque chose de plus personnalisable, via un menu d'options par exemple).
Nous allons commencer par détecter l'appui sur les touches. Attention ici, afin d'éviter des erreurs éventuelles, on va commencer à détecter l'appui sur une touche uniquement à partir du moment ou la page est totalement chargée.

Ajoutez donc le code suivant en bas, à l'intérieur de la fonction "windows.onload" :

Code : JavaScript

```
// Gestion du clavier
window.onkeydown = function(event) {
 alert('test');
 return false;
}
```

Vous pouvez tester, vous allez voir que dès que vous appuierez sur une touche du clavier, le message "test" apparaîtra.
Ici, j'ai retourné faux, tout simplement pour éviter que le comportement normal de la touche ne soit utilisé. Imaginez que vous ayez des barres de défilement sur votre page : à chaque fois que vous appuierez sur une flèche pour vous déplacer, elles vont bouger avec. En retournant faux, ce comportement ne se produira pas.

Maintenant, il va falloir identifier les touches pour produire une action en fonction de celle qui a été appuyée.
Pour cela, il faut d'abord récupérer l'objet correspondant à l'événement. Ensuite, il faut récupérer le code de la touche appuyée. Le code est soit dans l'attribut which, soit dans l'attribut keyCode (la plupart du temps keyCode fonctionne, mais sous Firefox par exemple ce n'est pas le cas lorsqu'on appuie sur une lettre) :

Code : JavaScript

```
var e = event || window.event;
var key = e.which || e.keyCode;
alert(key);
```

(pensez également à supprimer la ligne d'affichage du message de test, elle ne nous servira plus)

Maintenant, à chaque appui sur une touche, un nombre correspondant à cette touche s'affichera. Par exemple nous avons :

Touche	Code
Haut	38
Bas	40
Gauche	37
Droite	39
Entrée	13

Et on les trouve où ces codes ?

Je ne connais aucune documentation complète à ce sujet. Pour les trouver, j'utilise un alert, comme dans cet exemple.

Le reste du travail est assez simple, il suffit de tester le code et de déplacer notre personnage à gauche si le code de la touche est 37, à droite si le code est 39 ... Notez également que j'ai pensé aux joueurs qui préfèrent la combinaison ZQSD ou encore WASD à l'usage des flèches directionnelles  :

Code : JavaScript

```
switch(key) {
 case 38 : case 122 : case 119 : case 90 : case 87 : // Flèche haut,
z, w, Z, W
  joueur.deplacer(DIRECTION.HAUT, map);
  break;
 case 40 : case 115 : case 83 : // Flèche bas, s, S
  joueur.deplacer(DIRECTION.BAS, map);
  break;
 case 37 : case 113 : case 97 : case 81 : case 65 : // Flèche
gauche, q, a, Q, A
  joueur.deplacer(DIRECTION.GAUCHE, map);
  break;
```

```
        break;
      case 39 : case 100 : case 68 : // Flèche droite, d, D
        joueur.deplacer(DIRECTION.DROITE, map);
        break;
      default :
        //alert(key);
        // Si la touche ne nous sert pas, nous n'avons aucune raison de
    bloquer son comportement normal.
        return true;
    }
```

Vous remarquerez que pour chaque lettre, il y a deux codes différents, ce qui veut dire qu'un 'a' minuscule est différent d'un 'A' majuscule.

Vous pouvez maintenant tester le jeu, appuyez sur les flèches et TADAAAA ... **Ça ne fonctionne pas !**

La boucle principale

Reprenons dans l'ordre les grandes phases de l'exécution de notre jeu :

- Chargement de la page. Pendant ce temps, on initialise nos objets et on charge nos images.
- Fin du chargement de la page
- On dessine notre carte
- On associe l'événement window.onkeypress à notre fonction de détection
- L'utilisateur appuie sur une touche, ce qui modifie ses coordonnées x et y

Vous ne trouvez pas qu'il manque quelque chose après cela ?

Notre carte n'est pas redessinée après l'appui sur une touche ! Ce qui veut dire que les coordonnées x et y du personnage sont bien modifiées en mémoire, mais qu'on ne peut pas le voir puisque nous n'avons dessiné qu'une seule image, et puis plus rien.

Nous allons donc utiliser un genre de "boucle principale" (vous avez peut-être déjà entendu parler de ce concept par ailleurs). La boucle principale, c'est une boucle qui tourne en tâche de fond et qui redessine l'écran en permanence pour le garder à jour. Une fois mise en place, il nous suffira donc de modifier les coordonnées du personnage en mémoire pour qu'il soit automatiquement redessiné à sa nouvelle place sur l'écran quelques fractions de secondes plus tard.

En réalité ici ce n'est pas une boucle que nous allons utiliser. Si nous faisions une boucle infinie en JavaScript, le navigateur resterait bloqué (et stopperait l'exécution du programme au bout d'un moment). Nous allons donc utiliser la fonction setInterval. Cette fonction prend deux paramètres :

- Un callback, c'est-à-dire une fonction (ou une chaîne contenant un code à exécuter, ce qui est moins propre et moins performant)
- Un nombre en millisecondes

Une fois l'appel à cette fonction effectué, le callback sera automatiquement appelé par JavaScript toutes les n millisecondes, n étant le nombre passé en second paramètre.

Le premier paramètre sera simple : Il s'agira d'une fonction faisant un appel à la méthode de dessin de notre map.

Le cerveau humain est capable de discerner entre 20 et 30 images par seconde. Je vais donc régler le second paramètre à 40 millisecondes (1000 / 40 = 25 images par seconde, nous n'avons pas vraiment d'animation pour le moment, donc difficile d'évaluer avec précision ce que ça donnera, mais nous re-règlerons ce chiffre plus tard si l'animation n'est pas assez fluide).

Remplacez donc la ligne suivante de notre fichier :

Code : JavaScript

```
map.dessinerMap(ctx);
```

par ceci :

Code : JavaScript

```
setInterval(function() {
 map.dessinerMap(ctx);
}, 40);
```

Animation des déplacements

Nous allons maintenant nous attaquer à l'animation des personnages lors de leurs déplacements. Nous n'aurons besoin ici de ne modifier que la classe Personnage.

La mise en place de l'animation se déroulera en deux parties principales :

- L'animation elle-même (c'est-à-dire qu'on fera en sorte que durant le déplacement, on voie les jambes du personnage bouger)
- La mise en place d'un mouvement fluide (au lieu que le personnage se téléporte d'une case à l'autre, il y passera progressivement).

L'animation elle-même

Nous avons vu que notre fichier de sprite contient, pour chaque direction, quatre images (frames) différentes.
Si on y regarde de plus près, voici l'organisation des frames (de gauche à droite) :

- La première image est celle où le personnage est immobile
- Sur la seconde image, le personnage avance son pied droit
- Sur la troisième, il a de nouveau les jambes parallèles (en fait, il s'est avancé sur son pied droit et s'apprête à avancer le gauche pour finir un pas)
- Sur la quatrième, il avance le pied gauche

Ce schéma se répète à l'infini, une fois arrivé à la quatrième image, on repart à la première. Le personnage se retrouve donc droit, prêt à faire un autre pas.

Pour réaliser cette animation, il va nous falloir deux paramètres :

- Un repère pour mesurer le temps
- La durée (dans l'unité choisie) entre deux frames pour réaliser cette animation

Pour le second, c'est assez facile. Nous allons déclarer une constante, à laquelle nous donnerons une valeur initiale. Une fois que l'animation fonctionnera, il nous suffira de modifier cette valeur pour avoir une animation à la bonne vitesse.

Pour le repère temporel, nous avons deux choix :

- Utiliser l'heure du système (c'est-à-dire le timestamp)
- Compter le nombre de passages dans la boucle principale, c'est-à-dire le nombre de fois qu'on redessine le personnage sur l'écran (qui est régulier).

J'ai choisi la deuxième option pour une raison principale : si nous souhaitons mettre un système de pause dans le jeu (plus tard), il nous sera impossible de mettre aussi l'horloge en pause  . Nous risquerions donc de rencontrer des décalages dans l'animation après les pauses (ce qui n'est pas un bug primordial, mais autant l'éviter autant que possible).

Pour la durée de chaque frame de notre animation, il nous faut simplement un entier. On peut donc déclarer cette constante (je la déclare juste après les constantes de direction) :

Code : JavaScript

```
var DUREE_ANIMATION = 4;
```

Ici, nous avons une durée de quatre. Cela signifie que chaque fois qu'on aura redessiné quatre fois un personnage en train de bouger, il faudra passer à la frame suivante de l'animation. J'ai choisi ce chiffre en testant cette partie une fois achevée, mais au départ j'avais mis un chiffre bien trop grand (tellement grand que le personnage semblait immobile ).

Nous allons avoir besoin de certaines informations pour gérer cette animation au niveau du personnage :

- L'état du personnage, c'est-à-dire si le personnage est immobile ou non
- Si le personnage est en mouvement, le nombre de passages effectué par la boucle principale, afin d'afficher la bonne image.

Je vais combiner ça en un seul attribut, qui contiendra :

- Un nombre négatif si le personnage est immobile (Nous utiliserons -1, mais il vaut mieux tester s'il est négatif plutôt que de vérifier s'il est exactement égal à -1. Ainsi, l'attribut peut être potentiellement réutilisable pour stocker des informations concernant l'animation).
- Un nombre positif ou nul si le personnage est en mouvement. Dans ce cas, ce nombre est le nombre de passages effectués dans la boucle principale.

Nous pouvons donc initialiser notre attribut à -1, pour que le personnage soit immobile quand il apparaît dans le jeu :

Code : JavaScript

```
this.etatAnimation = -1;
```

Au moment où le personnage commencera à se déplacer, il faudra démarrer l'animation, et donc mettre cet attribut à 1 (on pourrait aussi le mettre à 0, mais cela décale l'animation d'une frame et provoque un court arrêt de l'animation entre deux cases). Ajoutez donc ce code dans la méthode "deplacer", juste avant les deux lignes où les attributs x et y sont modifiés :

Code : JavaScript

```
// On commence l'animation
this.etatAnimation = 1;
```

Ce que nous sommes en train de mettre en place est étroitement lié avec la mise en place d'un mouvement plus fluide. Nous allons donc avoir une animation qui - pour le moment - ne s'arrêtera jamais à partir du moment où un premier déplacement a été effectué.

Ensuite, il faut calculer l'image à afficher en fonction de l'animation courante. Modifiez donc la méthode "dessinerPersonnage", et ajoutez ceci au début :

Code : JavaScript

```
var frame = 0;
if(this.etatAnimation >= 0) {
 frame = Math.floor(this.etatAnimation / DUREE_ANIMATION);
```

```
        if(frame > 3) {
          frame %= 4;
        }
        this.etatAnimation++;
      }
```

Par défaut, nous prenons donc l'image immobile, c'est-à-dire l'image 0 (il y a quatre images dans notre animation, que l'on compte de 0 à 3). Si l'animation est commencée, on calcule la bonne image à afficher (on compte le nombre total de "pas" effectués dans l'animation, c'est-dire le nombre de fois où l'on a changé d'image. Si le nombre est supérieur à trois (qui est le numéro correspondant à la dernière image de l'animation), l'image à prendre est ce nombre modulo quatre (le reste de la division de ce nombre par le nombre d'images dans l'animation).

Il ne nous reste plus qu'à utiliser ce nombre. Dans l'appel à la méthode "drawImage" de l'objet context, nous avions mis un paramètre à 0. Il nous suffit de le remplacer par le numéro de l'image que nous venons de calculer, multiplié par la largeur du personnage :

Code : JavaScript

```
this.largeur * frame
```

Mise en place d'un mouvement fluide

Pour que le déplacement se fasse de manière fluide, et pas par "téléportation", il nous faut tout d'abord définir une vitesse de déplacement. Ce nombre nous indiquera combien de passages par la boucle principale seront nécessaires pour que le personnage passe d'une case à l'autre (les deux cases étant adjacentes). Je vais donc définir une constante, juste à côté de celle que nous avons déclarée pour l'animation :

Code : JavaScript

```
var DUREE_DEPLACEMENT = 15;
```

Maintenant il faut définir un point important : lorsque le personnage se trouve entre deux cases, quelles sont ses coordonnées ?

- Soit les coordonnées qu'il avait avant de se déplacer
- Soit ses nouvelles coordonnées (on anticipe donc le déplacement)

J'ai choisi le deuxième cas pour que nous n'ayons pas de problèmes "d'accès concurrents" à l'avenir. Je m'explique : si nous avons un autre personnage à l'écran qui souhaite aller sur la même case que nous et que durant le déplacement nous sommes toujours (en mémoire) sur la case précédente, l'autre pourra venir sur cette case, et nous serons alors deux sur une même case. Or, dans un prochain chapitre (sur les obstacles), nous ferons en sorte qu'il ne puisse y avoir qu'un seul personnage par case ! De même, si on prend un peu le problème à l'envers, si un personnage souhaite aller sur la case dans laquelle nous nous trouvions avant le déplacement alors que nous sommes entre deux cases, il devra attendre que nous ayons terminé le déplacement dans le premier cas. Dans le deuxième cas, il pourra commencer son déplacement sans problème.

À partir du temps total nécessaire à un déplacement, et du temps passé depuis le début du déplacement (que nous avons déjà dans l'attribut "etatAnimation"), nous allons - à chaque dessin - pouvoir calculer quelle proportion (pourcentage) du déplacement a été effectuée (nous obtiendrons un nombre décimal entre 0 et 1) pour passer d'une case à l'autre. En multipliant ce chiffre par 32 (qui est la taille d'une case), nous obtiendrons le nombre de pixels de "décalage" à appliquer (c'est-à-dire à ajouter ou soustraire en fonction de la direction) à la position du personnage en pixels.

La première chose à laquelle il faut penser (car on l'oublie souvent, ce qui provoque des choses assez étranges, vous n'aurez qu'à commenter ce bloc quand nous aurons terminé, et vous verrez), c'est d'empêcher le personnage de se déplacer tant qu'il y a déjà un déplacement en cours. Pour cela, il nous suffit d'ajouter la ligne suivante au début de la fonction "deplacer" :

Code : JavaScript

```
// On ne peut pas se déplacer si un mouvement est déjà en cours !
if(this.etatAnimation >= 0) {
 return false;
}
```

Pour l'autre partie du code (que j'ai déjà décrit dans les grandes lignes), je vais vous donner directement le bloc de code (commenté) à remplacer (les modifications seraient un peu complexes à détailler ligne par ligne). Dans la fonction dessinerPersonnage, remplacez le code suivant (que nous avons défini juste avant) :

Code : JavaScript

```
var frame = 0;
if(this.etatAnimation >= 0) {
 frame = Math.floor(this.etatAnimation / DUREE_ANIMATION);
 if(frame > 3) {
  frame %= 4;
 }
 this.etatAnimation++;
}
```

par ceci :

Code : JavaScript

```
var frame = 0; // Numéro de l'image à prendre pour l'animation
var decalageX = 0, decalageY = 0; // Décalage à appliquer à la
position du personnage
if(this.etatAnimation >= DUREE_DEPLACEMENT) {
 // Si le déplacement a atteint ou dépassé le temps nécessaire pour
s'effectuer, on le termine
 this.etatAnimation = -1;
} else if(this.etatAnimation >= 0) {
 // On calcule l'image (frame) de l'animation à afficher
 frame = Math.floor(this.etatAnimation / DUREE_ANIMATION);
 if(frame > 3) {
  frame %= 4;
 }

 // Nombre de pixels restant à parcourir entre les deux cases
 var pixelsAParcourir = 32 - (32 * (this.etatAnimation /
DUREE_DEPLACEMENT));

 // À partir de ce nombre, on définit le décalage en x et y.
 // NOTE : Si vous connaissez une manière plus élégante que ces
quatre conditions, je suis preneur
 if(this.direction == DIRECTION.HAUT) {
  decalageY = pixelsAParcourir;
 } else if(this.direction == DIRECTION.BAS) {
  decalageY = -pixelsAParcourir;
 } else if(this.direction == DIRECTION.GAUCHE) {
  decalageX = pixelsAParcourir;
 } else if(this.direction == DIRECTION.DROITE) {
  decalageX = -pixelsAParcourir;
 }

 this.etatAnimation++;
}
/*
* Si aucune des deux conditions n'est vraie, c'est qu'on est
immobile,
```

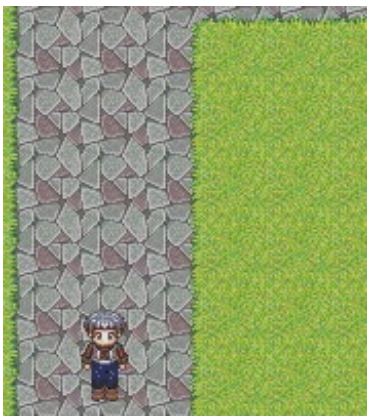
```
        * donc il nous suffit de garder les valeurs 0 pour les variables
        * frame, decalageX et decalageY
        */
```

Il ne nous reste plus qu'à additionner nos variables decalageX et decalageY aux paramètres passés à la méthode drawImage. Ce qui nous donne ceci pour les 6ème et 7ème paramètres :

Code : JavaScript

```
        // Point de destination (dépend de la taille du personnage)
        (this.x * 32) - (this.largeur / 2) + 16 + decalageX, (this.y * 32) -
        this.hauteur + 24 + decalageY,
```

Et voilà, si vous avez bien suivi les étapes, vous devriez pouvoir vous déplacer librement dans le jeu, ce qui devrait donner quelque chose comme ceci (ce gif est beaucoup moins beau et fluide que dans la réalité) :



Vous pouvez télécharger le code source de cette partie ici, ou bien le tester ici.

Nous sommes maintenant capables d'afficher une carte de n'importe quelle taille sur l'écran, ainsi que des personnages. Bien sûr notre jeu est loin d'être complet, mais ce premier résultat n'est-il pas satisfaisant ?