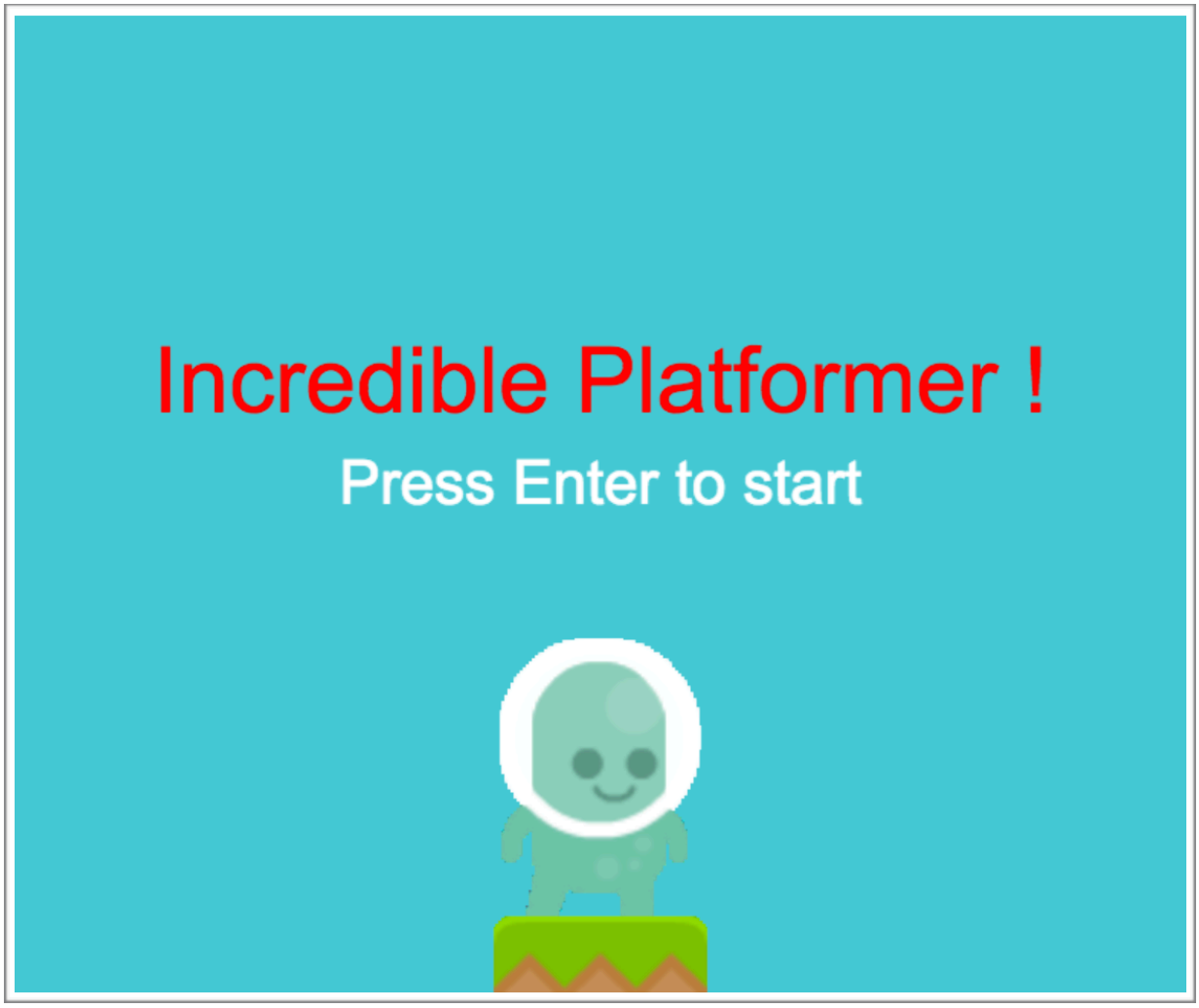
Projet PWA : Incredible Platformer



Ecran d'accueil du jeu

Rodolphe Cargnello

16 Avril 2017

Présentation du projet

Le but de ce projet est de concevoir un de plates-formes similaires à Mario en Javascript en utilisant les connaissances vu en cours tout au long de ce semestre. Pour pouvoir développer un tel jeu, il a fallu respecter quelques aspects de base à savoir:

- Le personnage doit pouvoir se déplacer horizontalement en marchant et verticalement en sautant ou en tombant.
- Le niveau doit comporter des plates-formes permettant d'éviter des obstacles (létales).
- Les niveaux doivent être représenté de manière générique dans un fichier de configuration au format JSON.
- Le jeu doit implémenter des mouvements réalistes (implémenter un moteur physique simpliste).
- Le rendu doit être fait dans un élément canvas.
- Le rendu doit être au minimum de 30 FPS, voir idéalement 60 FPS.

Une fois ces aspects respectés, le jeu doit aussi implémenter des fonctionnalités avancée. J'ai donc décidé de retenir ces trois fonctionnalités :

- Barre de vie et blessures : Le personnage dispose d'une barre de vie et puisse être blessé par des éléments du décor.
- Vitesse de course : on souhaite que sur pression simultanée d'un bouton en plus des touches de directions le personnage puisse courir.
- Jeu complet : le jeu doit être composé de plusieurs niveaux, s'enchaînant les uns avec les autres et d'un écran d'accueil. Il possède aussi une notion de « vies » qui permet au joueur de ne pas recommencer du début s'il/elle perd à un niveau donné.

Dans ce rapport, nous allons étudier certaines de ces fonctionnalités (de base et avancées) en examinant le choix et la manières dont ces dernières ont été implémentées.

Structure du projet

Afin de pouvoir implémenter ce jeu de plates-formes de manières structurée, j'ai décidé d'adopter un Design Pattern semblable au model MVC (Modèle Vue Controller)

Comme on peut le voir sur cette image, la partie modèle qui concerne les données du jeu ainsi que les différents algorithmes sont contenu dans un dossier nommé « engine » :
- constantes.js : Constantes du jeu (dimension des blocs)
- AABB.js : Algorithme de détection de collision
- vector.js : Classe provenant du « TP moteur physique » représentant un élément de 2 dimensions (posision, vitesse, force, etc)
- rect.js : Classe provenant du « TP moteur physique » représentant un élément un rectangle.
- entity.js : Classe héritant de « Rect » représentant un élément ayant des particularités physique.
- tile.js : Classe héritant de « Entity » représentant des éléments du décor.
- character.js : Classe héritant de « Entity » représentant le personnage contrôlé par le joueur
- world.js : Classe représentant le niveau dans son intégralité (le chargement des niveaux, l'interation du joueur avec les différentes entités, etc)

La partie controller est composée de 2 éléments :
- renderer.js : Classe permettant de dessiner les niveaux/écran d'accueil et de récupérer les interactions clavier du joueur.
- camera.js : Classe permettant de centrer la vue sur le personnage.

Ces deux dernière partie sont reliée par « main.js » qui effectue la connection entre elle, à savoir le chargement des niveaux et des images et le lancement du jeu.

La partie vue est représenté par la partie HTML et CSS dans les fichiers index.html qui contient nos différent canvas dans lesquels seront dessiné le jeu (cnvWorld pour dessiner le niveau, cnvChar pour dessiner le joueur et cnvGUI pour dessiner les éléments visuels relatifs aux messages et instructions du jeu).

Maintenant que nous avons vu la manière dont était organisé le code du projet, nous allons regarder de plus prêt la manière dont les données sont récupéré au lancement du jeu.

Chargement des données

Comme nous l'avons mentionné dans la section précédente, l'ensemble des donnés et préalablement chargé avant l'exécution du jeu dans le fichier « main.js » de la façon suivante :

```
"use strict";
window.addEventListener ("load", () =>
{
    let dataLvls = null;
    let dataAtlas = null;
    let playerAtlas = null;

    ...

    var start = Promise.resolve();
    start
        .then (loadLvls)// On charge le fichier contenant le niveau
        .then (loadAtlas)// L'emplacement des textures du monde
        .then (loadCharacterAtlas)// L'emplacement des textures du joueurs
        .then (loadImages)// On charge les fichiers images
        .then (startGame)// On charge les fichiers images
});
```

On utilise la fonctionnalité des Promesses qui permettent de pouvoir effectuer des traitement asynchrones sans pour autant bloqué la fenêtre sur une requête qui peut être longue, en

l'occurence si l'on veut charger des fichier images ou bien des fichiers de configuration, comme ce que l'on cherche à faire ici. De plus les promesses sont traité une par une, et lorsque l'on n'arrive pas à effectuer un traitement dans une promesse particulière, les suivantes ne seront pas exécutées. Il s'agit donc d'un moyen élégant et pratique pour effectuer nous traitements.

A présent, regardons plus attentivement l'une de ces promesses :

```
1  // On charge les fichiers contenant un niveau
2  function loadLvls (filename)
3  {
4      let lvls = ["lvl1", "lvl2", "lvl3"];
5      let lvlsPromises = lvls.map (function (lvl)
6      {
7          return  new Promise ( (ok, error) =>
8          {
9              let request = new XMLHttpRequest();
10             request.open("GET", "lvl/"+lvl+".json");
11             request.onreadystatechange = function ()
12             {
13                 if (request.readyState == 4)
14                 {
15                     if (request.status == 200)
16                     {
17                         ok (JSON.parse(request.responseText));
18                     }
19                     else
20                     {
21                         console.log(request.status);
22                         error ("chargement du fichier json");
23                     }
24                 }
25             };
26             request.send();
27         });
28     });
29     return (Promise.all(lvlsPromises));
30 }
```

Dans ce cas bien particulier, nous cherchons à charger plusieurs fichiers, en l'occurence les différents niveaux du jeu. Nous allons donc avoir recours à plusieurs traitement asynchrones lors ce l'exécution de cette fonction, autrement dit nous aurons besoin d'une promesse pour chacune d'entre-elles. Nous allons donc créer une listes de promesse en itérant sur notre liste de niveau à charger (l.5). On créé donc une nouvelle instance de l'objet « Promise » qui prend en paramètre « ok » qui contiendra nos données chargés, « error » qui contiendra le message d'erreur dans ce cas précis si la requêtes venait à échouer. Dans cette promesse on effectue une requête pour pouvoir charger le fichier en question et ajouter sont contenu dans « ok ».

Ainsi la fonctions « loadLvls » pour retourner l'ensemble de promesses que contient « lvlsPromises », ce qui permettra à la fonction suivante de récupérer tous nos niveaux.

La structure d'un niveau se présente de la façon suivante :

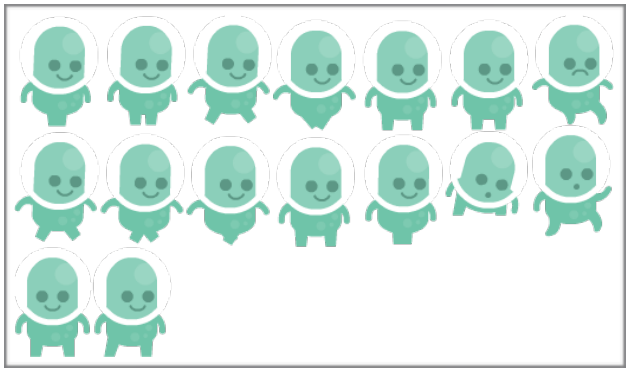
```
{
    "cases" :
    {
        "s" : {"texture": "stoneCenter", "traversable": false, "letal" : false},
        "g" : {"texture": "grassMid", "traversable": false, "letal" : false},
        "c" : {"texture": "castleMid", "traversable": false, "letal" : false},
        "C" : {"texture": "castleCenter", "traversable": false, "letal" : false},
        "L" : {"texture": "liquidLava", "traversable": false, "letal" : true}
    },
    "world":
    {
        "width": 45,
        "height": 16,
        "matrix":
        [
            "                                             ",
            "                                             ",
            "                                             ",
            "                                             ",
            "                                             ",
            "                                             ",
            "                                             ",
            "                                             ",
            "                                             ",
            "                                             ",
            "                                            @",
            "                                  ggggggggggg",
            "           cccc              g               ",
            "  X       CCCCCC       g                     ",
            "gggggg  gggggggggggggggg                     ",
            "ssssssLLssssssssssssssssssLLLLLLLLLLLLLLLLLLL"
        ]
    }
}
```

Chaque niveau possède un dictionnaire (nommé « cases ») permettant de représenter les propriétés de chacun de nos élément présent dans le niveau à savoir sa texture, si elle est traversable ou si cet élément est létal au contacte. A ces propriété sont associé une lettre qui nous permettra d'identifier une case dans la section suivante.

La section « world » va permettre d'éditer la forme que prendra notre niveau. Cette section possède 3 attributs :

- « width » : Représente la taille en nombre de bloc sur la largeur de notre niveau
- « height » : Représente la taille en nombre de bloc sur la hauteur de notre niveau
- « matrix » : Représente la disposition des éléments dans notre niveau. Chaque ligne est représenté par une chaine de caractère de taille « width » dans laquelle son placé les cases mentionné dans le dictionnaire. Un espace représente le fait qu'il n'y ai pas de case, le caractère X représente la position initial du joueur et le @ le bloc de fin du niveau. Ces deux caractères sont réservé et donc il n'ai pas possible d'associer des propriété dans le dictionnaire à ces lettres. De plus il doit y avoir exactement autant de ligne que spécifié dans « height » sinon le chargement du niveau échouera.

Pour ce qui est du chargement d'images, j'ai choisi d'utilisé ce que l'on appelle des « sprite sheet ». Il s'agit d'une image regroupant plusieurs motifs organisé de la façon suivante :



```
{
  "player_frames" :
  [
    {"name" : "duck", "x" : 365, "y" : 98, "width" : 69, "height" : 71},
    {"name" : "front", "x" : 0, "y" : 196, "width" : 66, "height" : 92},
    {"name" : "hurt", "x" : 438, "y" : 0, "width" : 69, "height" : 92},
    {"name" : "jump", "x" : 438, "y" : 93, "width" : 67, "height" : 94},
    {"name" : "stand", "x" : 67, "y" : 196, "width" : 66, "height" : 92},
    {"name" : "walk0", "x" : 0, "y" : 0, "width" : 72, "height" : 97},
    {"name" : "walk1", "x" : 73, "y" : 0, "width" : 72, "height" : 97},
    {"name" : "walk2", "x" : 146, "y" : 0, "width" : 72, "height" : 97},
    {"name" : "walk3", "x" : 0, "y" : 98, "width" : 72, "height" : 97},
    {"name" : "walk4", "x" : 73, "y" : 98, "width" : 72, "height" : 97},
    {"name" : "walk5", "x" : 146, "y" : 98, "width" : 72, "height" : 97},
    {"name" : "walk6", "x" : 219, "y" : 0, "width" : 72, "height" : 97},
    {"name" : "walk7", "x" : 292, "y" : 0, "width" : 72, "height" : 97},
    {"name" : "walk8", "x" : 219, "y" : 98, "width" : 72, "height" : 97},
    {"name" : "walk9", "x" : 365, "y" : 0, "width" : 72, "height" : 97},
    {"name" : "walk10", "x" : 292, "y" : 98, "width" : 72, "height" : 97}
  ]
}
```

On dispose dans cette image un ensemble de « sous-images » représentant ici le personnage et ses différents états. Pour pouvoir récupérer chacune des régions, on utilise un fichier de configuration (dans ce cas un fichier JSON) permettant d'extraire la région voulu. L'avantage C'est qu'au lieu de charger N fichiers images, on en charge plus que 2. On utilise le même procédé pour les bloc des plate-formes à l'exception prêt qu'il s'agit d'un fichier xml pour les positions des régions (choix du créateur des images).

A cette étape nous avons toutes nos ressources qui sont chargées. On peut donc créer notre « world » qui contiendra nos niveaux, ainsi que notre « Renderer » qui pourra le monde créé ainsi que les images que nous avons préalablement chargé.

```
// Lancement du jeu
function startGame(images)
{
    try
    {
        let world = new World(dataLvls);
        let render = new Renderer(world, dataAtlas, playerAtlas, images);
        render.start();
    }
    catch (e)
    {
        console.log(e);
    }
}
```

Fonctionnement de World

Nous allons à présent nous intéresse au fonctionnement de la classe World.

Comme nous l'avons vu précédemment, on donne la liste des niveaux à charger a l'objet world lors de sa création.

```
class World
{
    constructor(jsonFileList)
    {
        this.game_over = false;
        this.lvls = jsonFileList;
        this.currentLvlIndex = -1;
        this.height = 0;
        this.width = 0;
        this.loaded = false;
        this.loadNextLvl();

        this.lastHit = 1;
    }
    ...
};
```

On garde en mémoire la liste de niveau et on fait appel à la méthode « loadNextLvl » pour pouvoir charger le monde.

```
for (let i = 0; i < this.height; i++) // Pour chaque ligne
{
    this.matrix[i] = [];
    ...
    for (let j = 0; j < this.width; j++) // Pour chaque cases
    {
        if(currentLvl.world.matrix[i][j] == 'X') // S'il s'agit du joueur
        {
            this.spawn = new Vector(j*GLOBAL.BSIZE.x,i*GLOBAL.BSIZE.y);
            this.player = new Character(this.spawn);
        }
        else if(currentLvl.world.matrix[i][j] == '@') // S'il s'agit de la fin du niveau
        {
            this.matrix[i][j] = new Tile(new Vector(j*GLOBAL.BSIZE.x, i*GLOBAL.BSIZE.y), "signExit", true);
            this.exit =  {"x": j, "y" : i};
        }
        else if(currentLvl.world.matrix[i][j] != ' ') // Si ça n'ai pas un espace vide
        {
            let traversable = cases[currentLvl.world.matrix[i][j]].traversable || false;
            let letal = cases[currentLvl.world.matrix[i][j]].letal || false;
            let texture = cases[currentLvl.world.matrix[i][j]].texture;
            this.matrix[i][j] = new Tile(new Vector(j*GLOBAL.BSIZE.x, i*GLOBAL.BSIZE.y), texture, traversable, letal);
        }
        else // Sinon on met la case à undefined
        {
            this.matrix[i][j] = undefined;
        }
    }
}
```

On parcourt la matrice du niveau courant et on initialise notre niveau avec une position de début et d'arrivé et des plates-formes.

L'objet world va aussi gérer à chaque pas de temps l'états du joueur et ses interactions ave les autres blocs.

```
1  // Mise à jour de l'état du monde
2  step(delta)
3  {
4      ...
5      // On centre la position du joueur
6      let posx = Math.floor((body.origin.x+GLOBAL.BSIZE.x/2)/GLOBAL.BSIZE.x);
7      let posy = Math.floor((body.origin.y+GLOBAL.BSIZE.y/2)/GLOBAL.BSIZE.y);
8      ...
9
10     // On ajoute la gravité (TP moteur physique)
11     ...
12     // On teste la collision sur un rayon de 2 blocs autour du joueur
13     for (let i = posy-2 ; i < posy+2; ++i){
14         for (let j = posx-2; j < posx+2; ++j){
15             ...
16             let tile = that.matrix[i][j];
17             if(tile != undefined){ // S'il y a un bloc
18                 if(AABB(body, tile)){ // S'il y a collision
19                     // Si le bloc en question est la fin de niveau on quitte
20                     ...
21                     // Si le bloc est traversable on oublie
22                     ...
23                     // On repositionne le joueur
24                     ...
25                     // Si le bloc est létal on enlève une vie au joueur
26                     ...
27                 }
28             }
29         }
30     }
31     ...
32     //On met à jour sa position
33     body.move(body.velocity);
34
35     // On limite la position du joueur en x pour ne pas sortir du niveau
36     body.origin.x = Math.min(Math.max(0, body.origin.x), this.width*GLOBAL.BSIZE.x - GLOBAL.BSIZE.x);
37     return false;
38 }
```

Pour éviter de calculer la collision avec tous les blocs du niveau, on récupère la position du joueur et on décide de définir s'il y a une collision avec les blocs avoisinant le joueur. De cette façon on optimise considérablement les calcules. On teste dans un premier temps si le bloc existe et ensuite on effectue ou non les calcules de re-positionnement du joueur en fonction des propriétés d'un bloc. On peut ainsi mettre à jour la position du joueur (l.33) et on limite le déplacement en X du joueur en fonction de la dimension du niveau.

Fonctionnement de l'affichage et des saisies clavier

Avant de passer au rendu graphique dessiné par l'objet renderer, nous allons nous intéresser au fonctionnement de la gestion de la saisie du clavier.

```
1  var keyPressed = {};
2  ...
3  // Lancement du jeu
4  start (){
5      let that = this;
6      document.addEventListener('keydown', e => {keyPressed[e.keyCode] = true;});
7      document.addEventListener('keyup', e => {keyPressed[e.keyCode] = false;});
8
9      ...
10
11     // Gameloop
12     function gameloop(now){
13         ...
14     };
15
16     function mainMenu(now){
17         ...
18     };
19
20     requestAnimationFrame(mainMenu);
21 }
```

Pour pouvoir détecter si une touche est pressée ou non, nous ajoutons deux événements à la page html pour pouvoir mettre à jour notre dictionnaire. En effet la variable « keyPressed » (l. 1) va contenir les états des touches, autrement dis si nous faisons pression sur une touche en particulier. L'événement « keydown » va permettre de mettre la touche associée à Vrai dans le dictionnaire. A l'inverse l'événement « keyup » va permettre de remettre la touche détecté comme étant relâchée.

Ainsi nous pouvons étudier les déplacements du joueur par l'intermédiaire de la fonction input.

```
// Saisie clavier
inputs()
{
    let player = this.world.player;
    let friction  = 0.35;
    let speedIncr = 0.07;

    // Si on appuis sur la touche de sprint Alt
    if (keyPressed[18]){
        // Si on appuis sur une touche de déplacement Left, Right, Q ou D
        // On met une plus grande vitesse au joueur en Vx
        if(keyPressed[68] || keyPressed[39]){//On augmente la vitesse progressivement
            player.velocity.x = Math.max(5, player.velocity.x);
            player.velocity.x += speedIncr;
            player.velocity.x = Math.min(10,player.velocity.x)
        }
        else if (keyPressed[81] || keyPressed[37]){
            player.velocity.x = Math.min(-5, player.velocity.x);
            player.velocity.x -= speedIncr;
            player.velocity.x = Math.max(-10,player.velocity.x)
        }
    }
    else{// Sinon ...
        // on met une vitesse plus faible au joueur en Vx
        if(keyPressed[68] || keyPressed[39]){
            player.velocity.x = 5;
        }
        else if (keyPressed[81] || keyPressed[37]){
            player.velocity.x = -5;
        }
    }

    // Si on reste immobile et qu'on ne saute pas -> Vx = 0
    if((!keyPressed[68] && !keyPressed[39]) && (!keyPressed[81] && !keyPressed[37])){
        if (!player.is_jumping){//On diminue la vitesse progressivement
            if(player.velocity.x > 0){
                player.velocity.x = Math.max(0, player.velocity.x - friction);
            }
            else if (player.velocity.x < 0){
                player.velocity.x = Math.min(0, player.velocity.x + friction);
            }
            else{
                player.velocity.x = 0;
            }
        }
    }

    // Si on appuis sur une touche de saut Up ou Z
    if(keyPressed[90] || keyPressed[38]){
        // On augmente la vitesse en Vy si on ne saute pas déjà
        if(!player.is_jumping){
            player.is_jumping = true;
            player.velocity.y = -10;
        }
    }
}

```

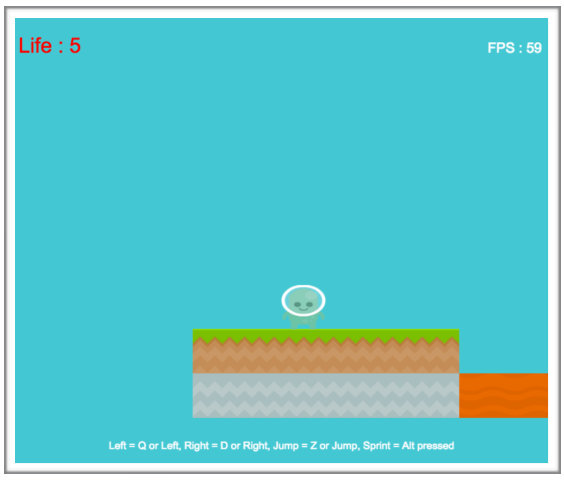
Grace à la technique du dictionnaire il est facile de tester si plusieurs touches sont pressée (dans le cas où l'on souhaite que le joueur accélère). Pour ce qui est de la vitesse de déplacement du joueur, on l'augmente progressivement en utilisant les fonction Math.min et Math.max de manière à ne pas dépasser un certain seuil => [-10;10].

Nous pouvons passer à la partie dessin du niveau.

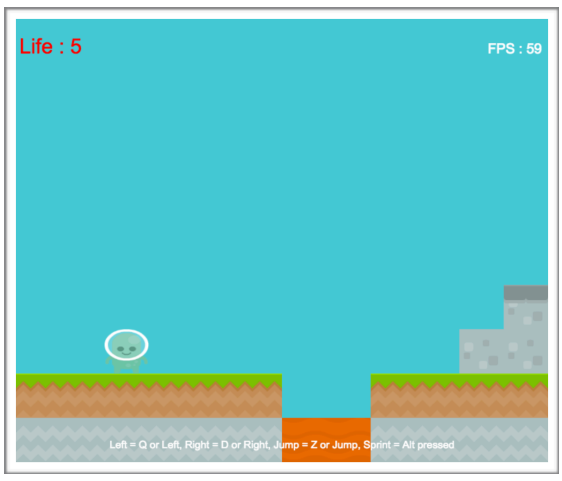
```
// On dessine la scène
draw(now, fps){
    this.ctxChar.clearRect(0,0,this.cnvWorld.width,this.cnvWorld.height);
    this.ctxWorld.clearRect(0,0,this.cnvWorld.width,this.cnvWorld.height);
    this.ctxGUI.clearRect(0,0,this.cnvWorld.width,this.cnvWorld.height);

    let player = this.world.player;

    // On met à jour la camera en fonction de la position du joueur
    let posx = Math.min(Math.max(768 - (768 - player.origin.x  + (player.width/2)), 768/2), this.world.width*GLOBAL.BSIZE.x - 768/2);
    let posy = Math.max(Math.min(player.origin.y  + (player.height/2), this.world.height*GLOBAL.BSIZE.y - 640/2 + GLOBAL.BSIZE.y), 640/2);
    this.camera.update(posx,posy);

    // On dessine une couleur de font
    this.ctxWorld.fillRect(0, 0, this.camera.screen.x, this.camera.screen.y);

    // On dessine toutes les tiles visibles par la camera
    for(let y = this.camera.startTile.y; y <= this.camera.endTile.y; ++y){
        for(let x = this.camera.startTile.x; x <= this.camera.endTile.x; ++x){
            if(this.world.matrix[y][x] != undefined){
                ...
                //On dessine le bloc à la position
                // x = this.camera.offset.x + (x*GLOBAL.BSIZE.x)
                // y = this.camera.offset.y + (tile.origin.y+GLOBAL.BSIZE.y)
            }
        }
    }
    ...

    // On dessine le joueur à la position
    // x = this.camera.offset.x + player.origin.x
    // y = this.camera.offset.y + (player.origin.y+GLOBAL.BSIZE.y)
    ...

    //GUI
    ...
}
```

On utilise 3 canvas pour pouvoir dessiner le rendu final. On utilise un canvas pour dessiner le personnage, un autre pour dessiner le monde et un dernier pour afficher l'interface du jeu. Il est inutile de dessiner ce qu'on ne vois pas en dehors du canvas, ainsi on utilise l'objet « camera » pour savoir quelle région on doit dessiner. Ainsi on dessine le canvas à partir de « camera.startTile » qui correspond au bloc en haut à gauche de l'écran, jusqu'à « camera.endTile » qui se situe en bas à gauche de l'écran.

De ce fait, on se doit de donner de bonnes valeurs à la caméra pour qu'elle se mettent à jour. En effet lorsque l'on se trouve en bordure du niveau, on ne souhaite pas voir l'extérieur :



Sans l'ajustement de la position

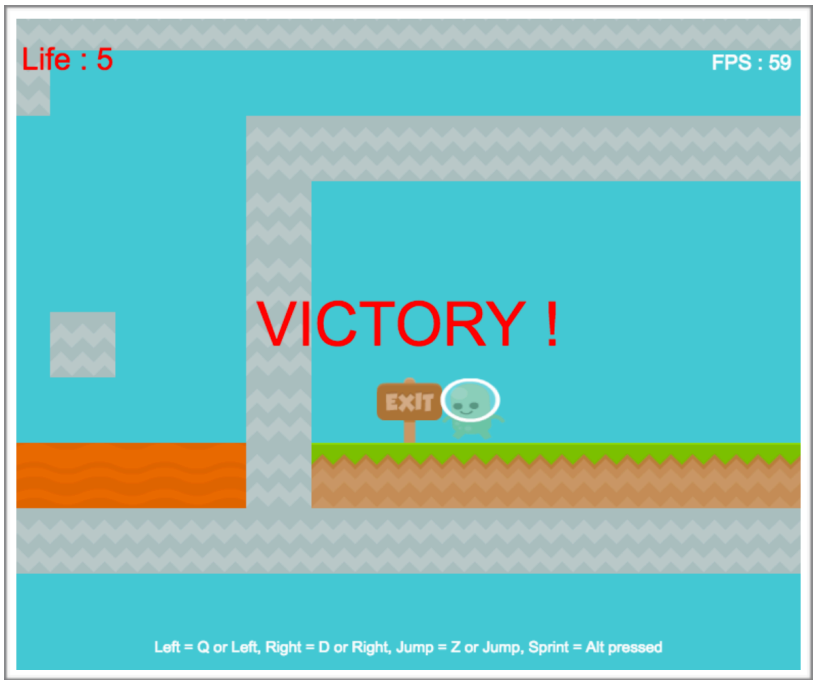


Avec l'ajustement de la position

On reprend le même principe que lorsque l'on restreignait la position du joueur pour pas qu'il puisse sortir du niveau.

Conclusion

`



A travers ce projet, j'ai pu mettre en avant la plupart des connaissances étudiées dans le module Programmation Web Avancé. Le jeu reste tout de même assez minimaliste en ce qui concerne sont contenu. Il n'y a pas d'ennemies, de blocs à formes complexes et la physique du jeu reste encore imparfaite. J'envisage de poursuivre le développement de ce projet en implémentant les autres fonctionnalités avancées.

Références

Documentation du langage : http://devdocs.io/javascript/

Moteur Physique : https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-the-basics-and-impulse-resolution--gamedev-6331

Viewport : http://technologies4.me/articles/viewport-culling-tile-map-a3/