Vincent Joulain

Projet Three JS

Subway runners

Sommaire

1) Cahier des charges

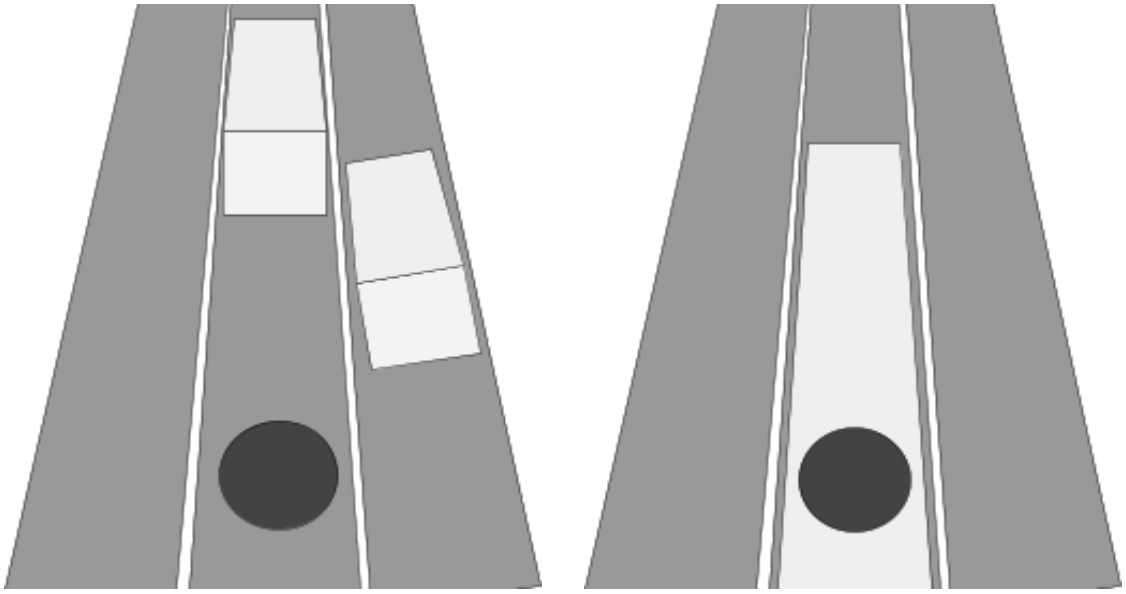
a) Présentation du projet

L'objectif de ce projet est de réaliser une version simplifiée du jeu très connu dans les années 2010, **Subway Surfers.** Le principe est très simple, courir de plus en plus vite en esquivant des obstacles tels que des trains à toute allure par des sauts et des roulades.

Il nous est aussi possible de ramasser des pièces (utiles que pour obtenir de nouveaux cosmétiques, le score dépendant de la distance parcourue) ou des objets spéciaux afin de faire évoluer le score plus rapidement.


La finalité de ce projet serait de s'approcher au maximum du gameplay original en utilisant des formes 3D simples, c'est-à-dire un objet géométrique représentant le personnage avançant en continu et esquivant d'autres objets 3D (tout d'abord statiques puis par la suite en mouvement) et, si le temps le permet, d'affiner ces formes en objets concrets animés.

Ci-dessous une modélisation de ce à quoi pourrait ressembler le projet dans sa première version :



b) Tâches à réaliser

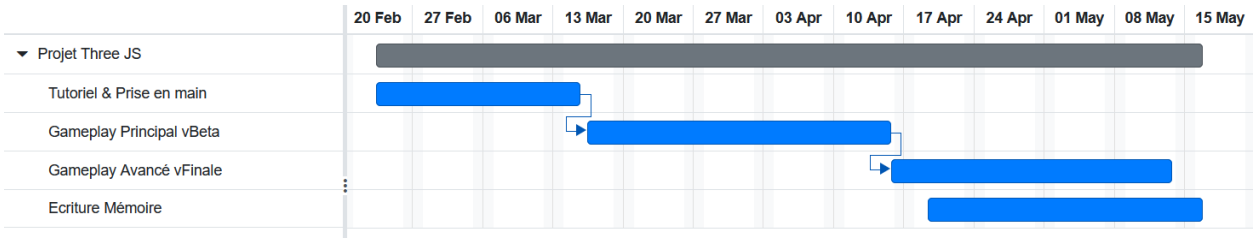
- Suivi du tutoriel provenant du site internet **Log Rocket** *(ressources 1°)* qui reprend les grandes lignes du gameplay recherché afin de prendre en main Three JS mais aussi d'avoir une base pour démarrer le projet.

- Développement du gameplay principal (version beta), implémentation des déplacements (mouvements droite gauche) et d'obstacles simples.

- Développement du gameplay avancé (version finale), création d'obstacles plus complexes (mise en mouvement), ajout du saut permettant au joueur de monter sur les obstacles, ajout de pièces à récupérer, si le temps le permet rendu plus esthétique du jeu.

- Rédaction du mémoire (10 - 15 pages) regroupant tous les éléments importants du projet (zones difficiles lors du développement, ressources …)

c) Calendrier



2) Premiers pas avec **Three.js**

a) Tutoriels et ressources utilisées

Comme précisé dans le cahier des charges, la première étape était de suivre le tutoriel **log rocket** *(ressources 1°)*, le projet présenté étant basé sur les mêmes principes de gameplay que le jeu **Subway Surfers,** ce tutoriel avait tout pour être une ressource solide à laquelle se référer durant toute la phase du développement du jeu. Toutefois, arrivé aux trois-quarts, cela ressemblait plus à une revue de code d'un projet fini et complexe plutôt qu'à un tutoriel. Rien d'intéressant n'était expliqué sur Three.js à part des choses très spécifiques ou avancées *(ex: rendu de textures animées – lac, ciel avec des nuages … –)*.

Je me suis donc redirigé sur le travail d'un des contributeurs de Three.js (*Lewy Blue*) qui a mis en ligne un manuel *(ressources 2°)* reprenant tous les grands concepts de la librairie, le format est plus celui d'un cours que d'un tutoriel mais il offre tout ce qui est nécessaire à la création et au développement d'un projet comme le mien.

J'ai eu l'occasion de compléter les 14 parties présentées dans ce manuel, d'abord les 9 premières avant de créer mon projet et par la suite les 5 suivantes au cours du développement en fonction de mes besoins.

b) Organisation de l’environnement de travail

Afin de pouvoir jouer à un jeu JavaScript sur un navigateur, il faut tout d’abord installer **node JS** *(version 18.2)* afin de pouvoir lancer un serveur local sur lequel faire tourner **Three.js**. Une fois node JS installé, il existe plusieurs moyens d’importer three.js; j’ai fait le choix de l’importer en tant que module et d’utiliser le bundler Webpack *(ressources 3°) (gestion des modules & ressources, minification du code …)*.

Une fois Webpack configuré et Three.js correctement importé, je suis passé à l'organisation du projet en lui même : dans le dossier **src/** on peut retrouver un sous-dossier au nom de **world/,** celui-ci contient l’entièreté du jeu et est complètement isolé du reste du code.

Il se sépare en **4 grandes parties** :

- **assets/** contient les modèles 3D importés ainsi que les textures
- **components/** contient tous les éléments faisant partie de la scène (objets, caméra, lumières …)
- **systems/** contient l’aspect dynamique du jeu, càd le moteur de rendu, la boucle d’animation ainsi que ce qui permet d’adapter la taille du rendu à la fenêtre de l’utilisateur
- **World.js** classe principale qui permet de créer et lancer le jeu depuis l'extérieur.

3) Développement du jeu

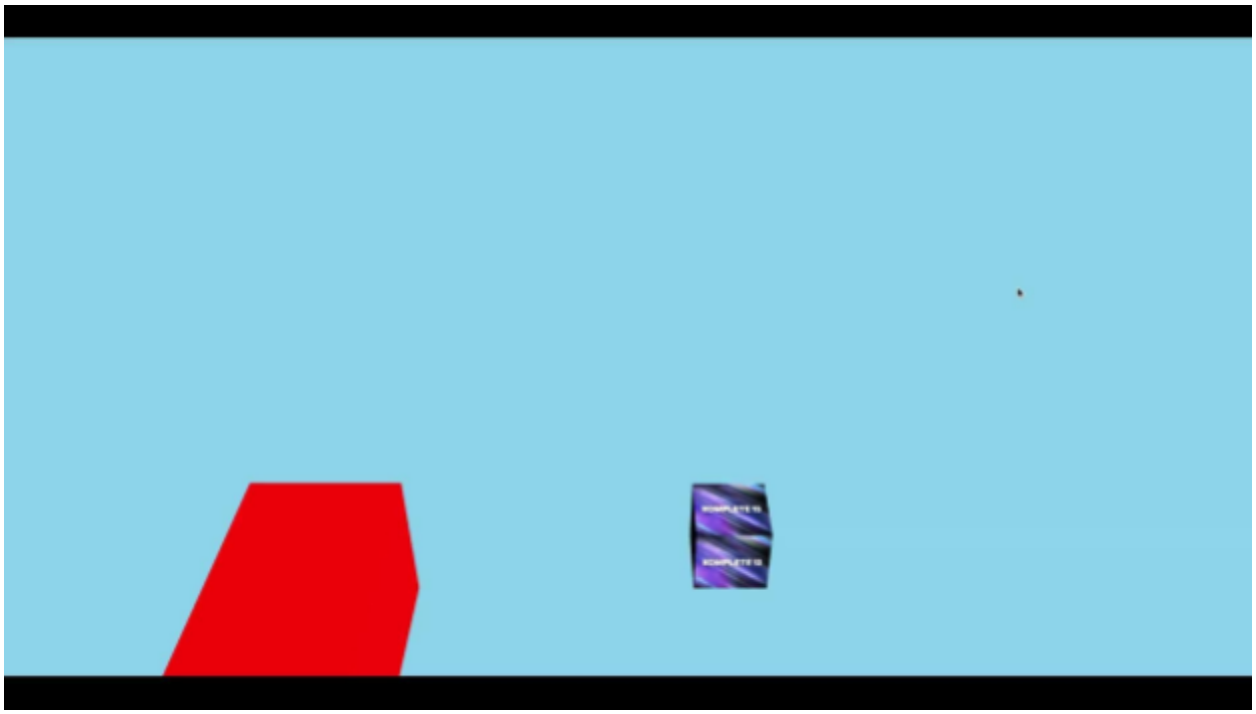
a) Première version : Mise en place du Gameplay

La première étape de ce projet est d'obtenir un gameplay fonctionnel avec des formes simples, c'est-à-dire créer un joueur *(cube)* capable de se déplacer sur trois grands axes, ceux-ci représentant les lignes imaginaires auxquelles il sera restreint. Une fois le joueur capable de se mouvoir sur les différentes droites *(droite, centre, gauche)* ainsi que de sauter et de se baisser nous pourrons passer à la création des ennemis.

L'ennemi prendra la forme d'un pavé droit *(représentant un train)* et devra avancer vers le joueur sur un des trois axes choisi aléatoirement à une vitesse prédéfinie *(susceptible d'augmenter au cours du jeu)*.

Les conditions de défaite sont la collision frontale et latérale avec l'ennemi, la collision du joueur avec la partie haute de l'ennemi *(toit du train)* doit lui permettre de courir dessus.

Ci-dessous une vidéo du rendu de la première version du jeu :



https://drive.google.com/file/d/1_up_DqlKil8WcLzztiHbRXQkiVDPYtya/view?usp=sharing

Afin d'obtenir ce résultat, il nous faut tout d'abord créer deux classes, **updatables/Player.js** et **updatables/Ennemi.js**, auxquelles nous devons assigner une forme 3D et un matériau :

Player.js : *(même processus pour Ennemi.js)*

```
class Player {
 constructor() {
   const geometry = new BoxBufferGeometry(1, 1, 1);
   const textureLoader = new TextureLoader();
   const texture = textureLoader.load(textureImage);
   const material = new MeshStandardMaterial({
     map: texture,
   });

   this.mesh = new Mesh(geometry, material);
```

Les deux objets créés et ajoutés à la scène, il nous faut maintenant les animer. Pour cela créons une autre classe **systems/Loop.js** qui va permettre de gérer l'animation de tous les objets présents sur la scène *(voir 1.7 The Animation Loop, manuel Lewi Blue, ressources 2°)*.

Le fonctionnement de la classe **Loop** est très simple, nous devons tout d'abord lui donner accès à la caméra, la scène ainsi qu'au renderer afin de lui permettre de mettre en place une boucle qui va demander un nouveau rendu à chaque nouvelle frame :

Loop.js :

```
 start() {
   this.renderer.setAnimationLoop(() => {
     this.tick();
     // render a frame
     this.renderer.render(this.scene, this.camera);
   });
 }
```

Comme il est possible de le voir ci-dessus, avant de faire un nouveau rendu, un appel à la fonction **tick** est effectué. Cette fonction est le cœur même de notre animation, c'est ici que chaque objet va être mis à jour.

Loop.js :

```
tick() {
  let delta = clock.getDelta();
  for (const object of this.updatables) {
      object.tick(delta);
  }
}
```

On s'aperçoit ici que cette méthode ne fait qu'appeler la fonction **tick** de chacuns des objets présents dans le tableau **updatables** *(contenant tous les objets de la scène pouvant êtres mis à jour)*. Cette fonction *(object.tick)* a pour objectif de déplacer l'objet dans l'espace avant que l'objet **Loop** puisse faire un nouveau rendu.

Sans rentrer dans les détails, la variable **delta** permet de normaliser les animations afin qu'elles ne dépendent plus du nombre d'actualisations par seconde *(ex: si la vitesse est mise à jour à chaque nouvelle frame, un ordinateur capable de 120 ips aura une animation deux fois plus rapide qu'un ordinateur capable de 60 ips).*

Exemple pour une rotation sur tous les axes :

```
player.tick = (delta) => {
  player.rotation.z += radiansPerSecond * delta;
  player.rotation.x += radiansPerSecond * delta;
  player.rotation.y += radiansPerSecond * delta;
};
```

Arrivés ici, nous avons un joueur et un ennemi, tous deux animés, ajoutés à notre scène. L'objectif suivant est la détection de collision. Une bonné idée serait d'utiliser les fonctions **intersects** et/ou **contains** de **three.js** qui permettent de détecter respectivement une collision entre une ou plusieurs surfaces ou une collision complète *(collision avec toutes les surfaces)*.

Toutefois, ces fonctions ne nous permettent pas de différencier les types de collisions *(latérales, frontales, par dessus / dessous)* or nous devons être capable de les discriminer car les actions du joueur ne sont pas les mêmes en fonction du type de collision.

Cependant tout n'est pas à refaire car si nous allons chercher la décomposition de ces fonctions sur le **GitHub** de **three.js** *(ressources 4°)* on retrouve :

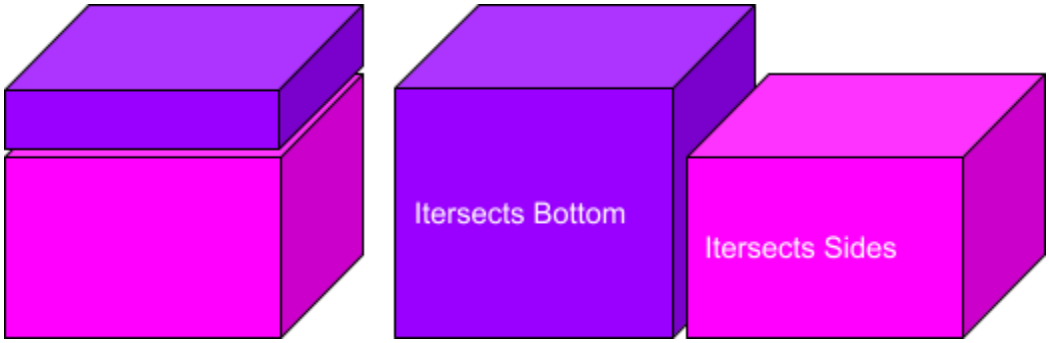
```
intersectsBox( box ) {
 // using 6 splitting planes to rule out intersections.
 return box.max.x < this.min.x || box.min.x > this.max.x ||
   box.max.y < this.min.y || box.min.y > this.max.y ||
   box.max.z < this.min.z || box.min.z > this.max.z ? false : true;
```

Etant donné la simplicité de la fonction, il nous est donc possible de créer nos propres fonctions de collision **intersects Sides** et **intersects Bottom** :

Player.js :

```
intersectsSides(ennemy) {
  if (
    this.boundingBox.max.x < ennemy.max.x &&
    this.boundingBox.min.x > ennemy.min.x
  ) {
    if (
      this.boundingBox.min.y + this.scale * 0.8 <= ennemy.max.y &&
      this.boundingBox.max.y >= ennemy.min.y
    ) {
      if (
        this.boundingBox.min.z < ennemy.max.z &&
        this.boundingBox.min.z > ennemy.min.z
      )
        return true;
    }
  }
  return false;
}
intersectsBottom(ennemy) {
  if (
    this.boundingBox.max.x < ennemy.max.x &&
    this.boundingBox.min.x > ennemy.min.x
  ) {
    if (
      this.boundingBox.min.y <= ennemy.max.y &&
      this.boundingBox.max.y >= ennemy.min.y
    ) {
      if (
        this.boundingBox.min.z < ennemy.max.z &&
        this.boundingBox.min.z > ennemy.min.z
      )
        return true;
    }
  }
  return false;
}
```

Les deux fonctions ci-dessus se comportent de manière très similaire, elles détectent toutes deux les collisions sur toute la surface sauf **intersects Sides** qui s'arrête à 80% de la hauteur :



il est intéressant de remarquer que **intersects Bottom** se comporte de la même façon que la fonction **intersects Box** interne a **three.js**, néanmoins la méthode avec laquelle les collisions allaient être détectés n'était pas encore définie au moment du développement, j'ai fait le choix de garder la fonction **intersects Bottom** pour plus de clarté *(et la capacité de la modifier si besoin par la suite)*.

b) Seconde version : Importation de modèles 3D & Animation

Le GamePlay mis en place, la seconde partie du projet consiste à importer / créer des objets 3D à ajouter au jeu afin de le rendre plus vivant. Nous avons besoin pour cela :

- d'un joueur
- d'un train
- de rails
- De décors (bâtiments, maisons, routes…)

Deux options s'offrent à nous, importer des modèles pré construits et animés dans un format facile à utiliser avec three.js ou bien construire nous même les modèles programmatiquement avec des formes simples ou complexes. La meilleure option étant toujours un entre-deux, le joueur et le train ont été importés, les rails et les décors construits.

i) Modèles Importés

La première chose à faire avant d'importer des modèles 3D est de choisir le format à utiliser et de, si possible, garder le même au sein de tout le projet afin d'obtenir un code homogène et facile à maintenir *(le format choisi est glb)*.

Le premier modèle importé est le joueur, il était nécessaire d'avoir un personnage animé afin d'obtenir un rendu visuel captivant. Il est issu des exemples du site officiel de **three.js** *(ressources 5°)*.

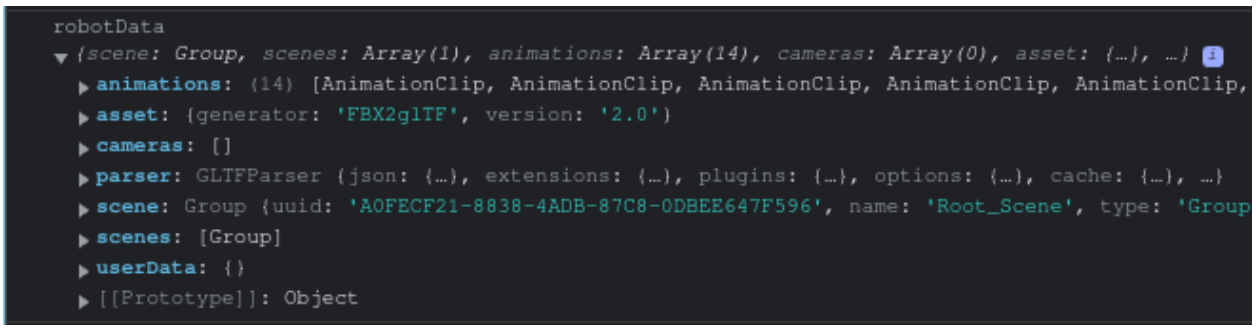


Afin de charger le modèle dans notre jeu il faut importer **GLTF Loader** depuis **three.js** ainsi que notre modèle 3D au format glb :

```
import { GLTFLoader } from "three/examples/jsm/loaders/GLTFLoader.js";
import robotModel from "./../../assets/models/RobotExpressive.glb";
```

Puis de charger *(de manière asynchrone)* le modèle lors de l'initialisation de notre objet :

```
async init() {
  const loader = new GLTFLoader();
  const robotData = await loader.loadAsync(robotModel);
```



Si on affiche le contenu de l'objet **robot Data** on peut s'apercevoir qu'il est composé de plusieurs parties, celles qui nous intéressent sont **animations** et **scenes**, la première contient les différentes animations que peut jouer notre personnage et la seconde la géométrie et les matériaux qui le composent.

On peut maintenant charger le corps et les animations de notre futur joueur :

```
  this.mesh = robotData.scene.children[0];
  this.mixer = new AnimationMixer(this.mesh);

  // load animations
  this.death = this.mixer.clipAction(robotData.animations[1]);
  this.jump = this.mixer.clipAction(robotData.animations[3]);
  this.run = this.mixer.clipAction(robotData.animations[6]);
```

On crée la hitbox :

```
  this.mesh.hitbox = new BoxBufferGeometry(1, 1, 1);
  this.mesh.hitbox.computeBoundingBox();

  this.boundingBox = new Box3(new Vector3(), new Vector3());
  this.boundingBox.setFromObject(this.mesh);
```

Notre personnage est maintenant prêt à être utilisé et importé dans notre jeu, il ne nous reste plus qu'à jouer les animations au bon moment. *(Modèle du Train importé de manière similaire)*

ii) Modèles Construits

Importer des modèles préconstruits ou modeler ses propres objets 3D grâce à des logiciels tels que Blender, 3Ds Max, Maya, etc, est la manière la plus efficace d'obtenir des formes complexes et animées. Toutefois il est possible de construire ses propres objets à l'aide des fonctions internes de three.js. Cette façon de modeler peut très vite devenir difficile à gérer, c'est pourquoi il est nécessaire de se cantonner à des objets simples avec peu de formes.

Cela convient parfaitement à la construction de rails ou de petites maisons.

Centrons nous sur la création d'une des trois différentes maisons présentes dans le jeu *(les rails étant moins intéressants car seulement composés de pavés droits de différentes tailles)* et décomposons sa fonction, celle qui à été choisie est la maison 1 *(la plus simple)* :

Il nous faut commencer par définir les dimensions, les valeurs choisies dépendent de l'angle de la caméra et du nombre de maisons qu'il à été décidé d'ajouter à la scène sur chaque côté de la route *(ici 3)* :

```
function createHouse1(position) {
 let dimension = { x: 5, y: 5, z: 14 };
 let hypotenus = Math.sqrt((dimension.x / 2) ** 2 * 2);
```

Une fois les dimensions correctes, nous devons passer à la création des différentes géométries et matériaux afin d'obtenir la forme 3D complète, par exemple pour la création du toit :

```
 const roofGeometry = new BoxBufferGeometry(
   hypotenus,
   hypotenus,
   dimension.z - 0.01
 );
const roofMaterial = new MeshStandardMaterial({ color: getRandRGB() });

let roof = new Mesh(roofGeometry, roofMaterial);
```

Il nous faut maintenant créer la maison et ajouter chacun de ses composants :

```
const house = new Group();
house.add(walls, roof, door, windowleft, windowright);
```

On termine par placer les différents éléments en fonction de leur position :

```
switch (position) {
   case "left":
     door.rotateY(MathUtils.degToRad(90));
     door.position.x = dimension.x / 2 + 0.01;
     windowleft.rotateY(MathUtils.degToRad(90));
     windowleft.position.x = dimension.x / 2 + 0.01;
     windowright.rotateY(MathUtils.degToRad(90));
     windowright.position.x = dimension.x / 2 + 0.01;
     break;
   case "right":
     door.rotateY(MathUtils.degToRad(270));
     door.position.x = -(dimension.x / 2 + 0.01);
     windowleft.rotateY(MathUtils.degToRad(270));
     windowleft.position.x = -(dimension.x / 2 + 0.01);
     windowright.rotateY(MathUtils.degToRad(270));
     windowright.position.x = -(dimension.x / 2 + 0.01);
     break;
 }
```

Notre maison vient d'être créée avec succès. Une fois toutes les maisons construites, divisées en deux groupes *(côté gauche et côté droit)* et placées sur la scène, nous devons compléter notre nouvel objet avec la fonction **tick** :

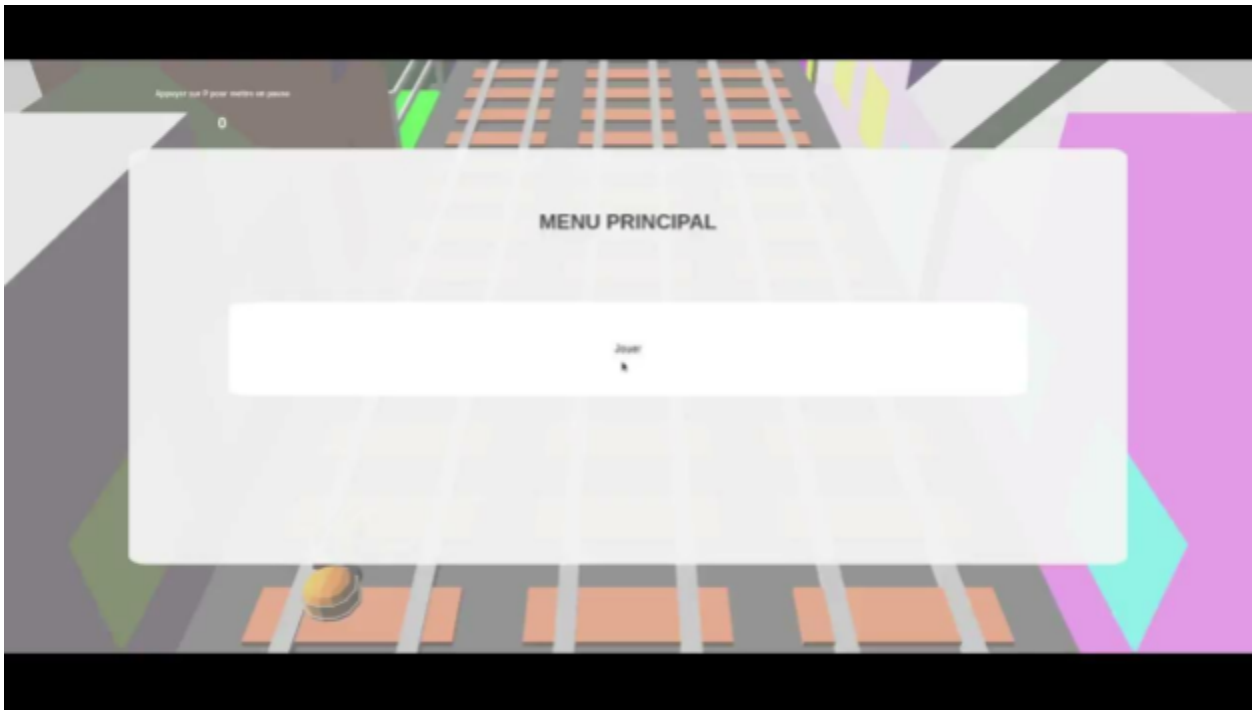
```
houses.tick = (delta) => {
   houses.children.forEach((house) => {
     house.position.z += 0.1;
     if (house.position.z >= 10) house.position.z = houseStartingZPos + 10;
   });
 };
```

Enfin, ajoutons chacun des groupes de maisons dans le tableau **updatables** de l'objet **Loop** précédemment créé :

World.js :

```
this.#loop.updatables.push(housesRight);
this.#loop.updatables.push(housesLeft);
```

Ci-dessous une vidéo du rendu de la seconde version du jeu :



https://drive.google.com/file/d/1Oi1vVFUMCiixbnU4wluHsHAy9p8hzurt/view?usp=sharing

c) Interface utilisateur

L’interface utilisateur est très simple. Elle est composée uniquement de trois menus ainsi que d’un affichage en cours de jeu précisant le score et l’ouverture du menu pause *(utilisation de la touche P)*.

Le score croît de manière exponentielle jusqu'à atteindre une limite au moment où le jeu atteint le maximum de sa difficulté.

4) Conclusion

Ce projet fut très intéressant dans son ensemble et m'a permis d'en apprendre beaucoup sur **three.js** ainsi que d'approfondir mes connaissances sur le langage JavaScript qui peut parfois être très permissif. La logique du jeu n'était rien de vraiment nouveau car elle ressemblait beaucoup à ce que nous avions déjà eu l'occasion de faire ce premier semestre et l'an passé avec Processing.

Ayant développé quelques jeux 3D (fps) ou 2D (échecs) avec des moteurs de jeux tels que **Unity** ou **Unreal Engine** par le passé, j'ai compris durant ce projet à quel point ils pouvaient s'avérer utiles sur beaucoups d'aspects lors de la création d'un jeu vidéo (gestion des rendus, des collisions, adaptabilité des formats / plateformes …)

Je suis satisfait du projet et suis arrivé à un rendu plus intéressant que celui que je prévoyais sur l'aspect visuel du jeu, toutefois je n'ai pas implémenté les objets collectibles et la duplication d'ennemis respectivement car j'ai préféré me concentrer sur d'autres aspects du jeu et parce que j'ai eu quelques soucis lorsque j'ai essayé d'ajouter la fonctionnalité *(qui semblaient être un gouffre à temps)*. Ce sont des choix que j'ai fait durant la phase de développement mais qui font de mon projet un jeu moins prenant que la version originale.

5) Ressources

1° LogRocket - Creating a game in Three.js :
https://blog.logrocket.com/creating-game-three-js/

2° Manuel Par Lewie Blue
https://discoverthreejs.com/book/

3° Webpack doc
https://webpack.js.org/guides/getting-started/

4° Décomposition de la fonction Intersect
https://github.com/mrdoob/three.js/blob/dev/src/math/Box3.js#L319

5° Modèle 3D du personnage
https://threejs.org/examples/#webgl_animation_skinning_morph