Rapport Final

du

Projet de L3

GAME FACTORY

Réalise par :

Simon ARNOULT
Wissame MEKHILEF
Vincent RENARD

Table des matières

1 Introduction

Sans avoir à rappeler le sujet, nous avons choisit de modéliser un monde 2D avec une vue lateral, les mouvements de caméra au n'était pas clairement définie au début du projet.
Quant au gameplay, on voulais un jeu simple à jouer avec une idée de emprunter au jeu de runner ou seul le saut est possible.

blabla bla

2 Présentation globale

2.1 Diagramme de classe

Le diagramme de classe ci-dessous donne un aperçu de l'état des classes, de leur association mais aussi de qui créer chacun des objets.
On se rend compte que WorldReader cree tout ce qui est nécessaire pour le monde.

2.2 Diagramme des boucles de jeu

2.2.1 Les contextes

Les contextes sont un concept que l'on a utilisé pour déterminer les états et un ainsi clairement séparé les moments dans le jeu pour ne faire les update et les rendue que des élements nécessaire.
De plus il nous ont permis de définir des règles pour passer d'un contexte a un autre.

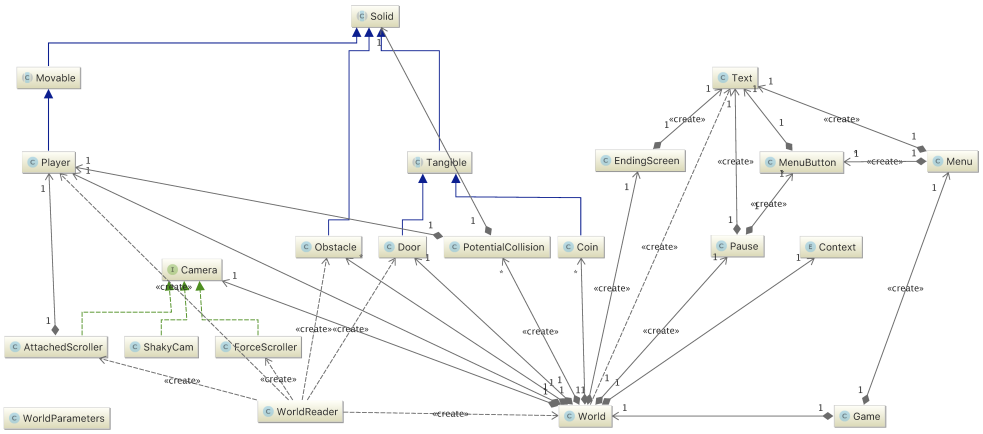

FIGURE 1 – Le diagramme de classe UML

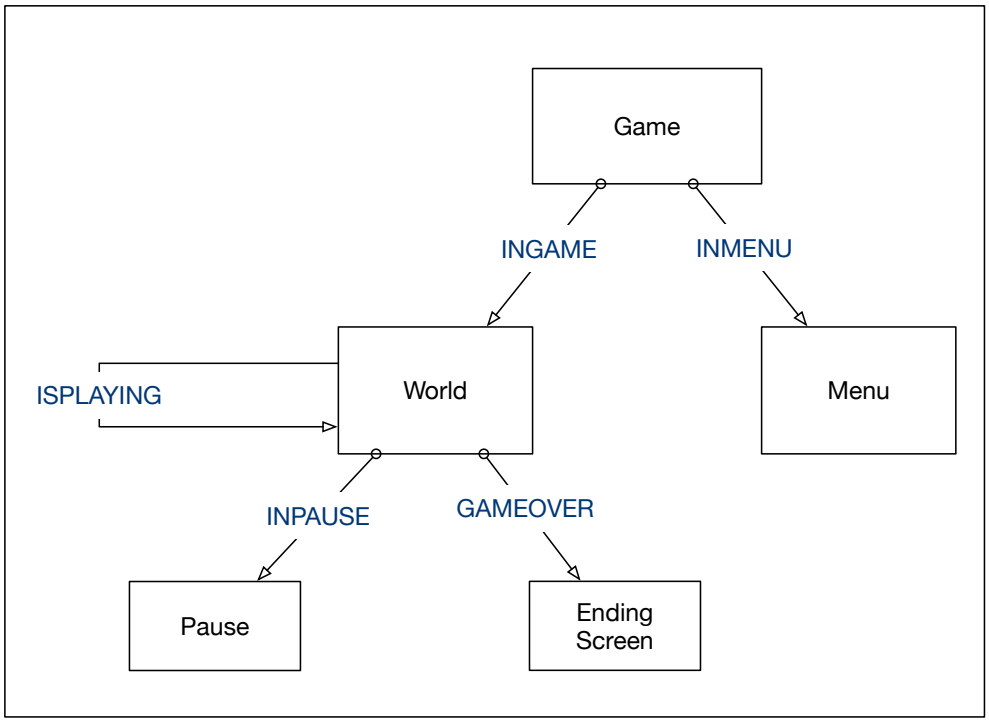

FIGURE 2 – schema 1

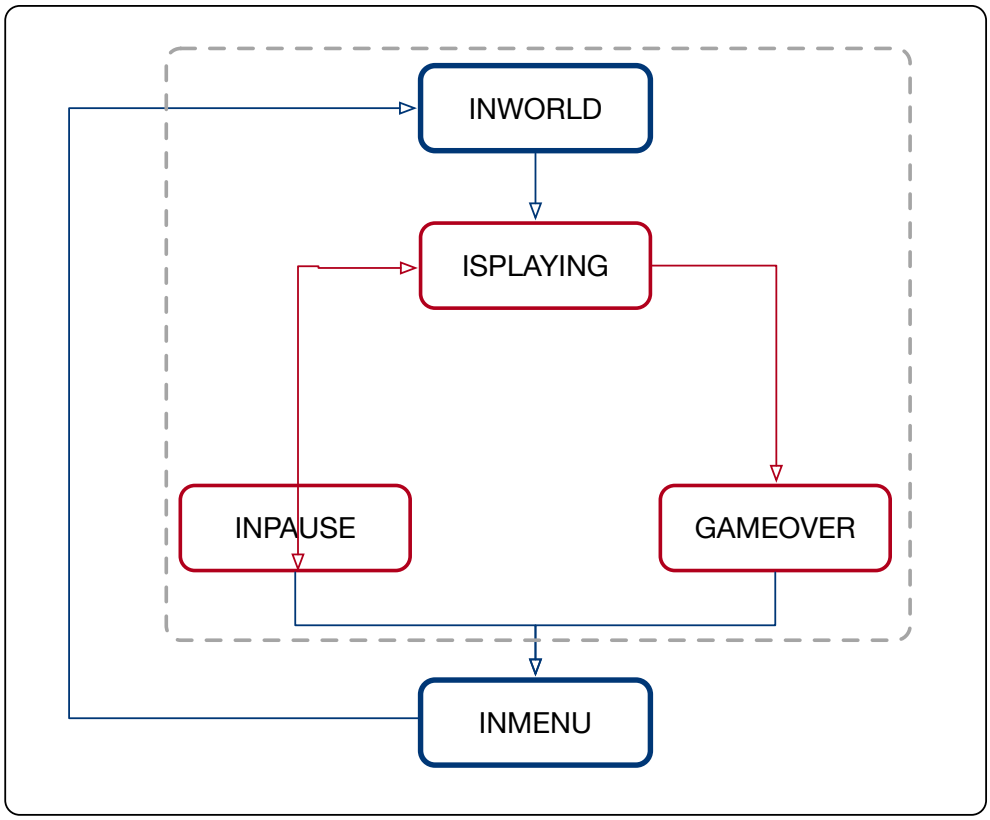

FIGURE 3 – schema 2

2.2.2 La boucle update

2.2.3 La boucle render

2.3 Architecture MVC

2.3.1 Affichage des données

2.3.2 Interpretation des entrées clavier

3 Une organisation sans faille

On a toujours durant le projet avancer par étapes nécessaire pour ne pas se perdre.

3.1 Description des taches

3.1.1 Recherche d'un framework + suivi d'un tutoriel

Nous avons choisi d'utiliser la librairie Lightweight Java Game Library (LWJGL) afin de gérer l'affiche graphique de notre jeu. Nous avons ensuite suivi le premier tiers du tutoriel " Jeu 2D avec LWJGL " sur la chaîne YouTube " Tuto Programmation (Marccspro) " afin de nous familiariser avec cette librairie.

3.1.2 Gestion d'un mouvement de caméra simple et autonome

Dans l'optique de pouvoir évoluer dans un monde plus grand que la fenêtre du jeu, nous avons implémenté un scrolling horizontal à vitesse fixée.

3.1.3 Gestion de l'interaction entre le joueur et la fenêtre

À ce stade du projet, nous avons créé un dépôt Git afin de mettre en commun notre travail, l'objectif à court terme étant de gérer simultanément la physique du joueur et le scrolling de la fenêtre, ainsi que les collisions entre les bords de cette dernière et le joueur.

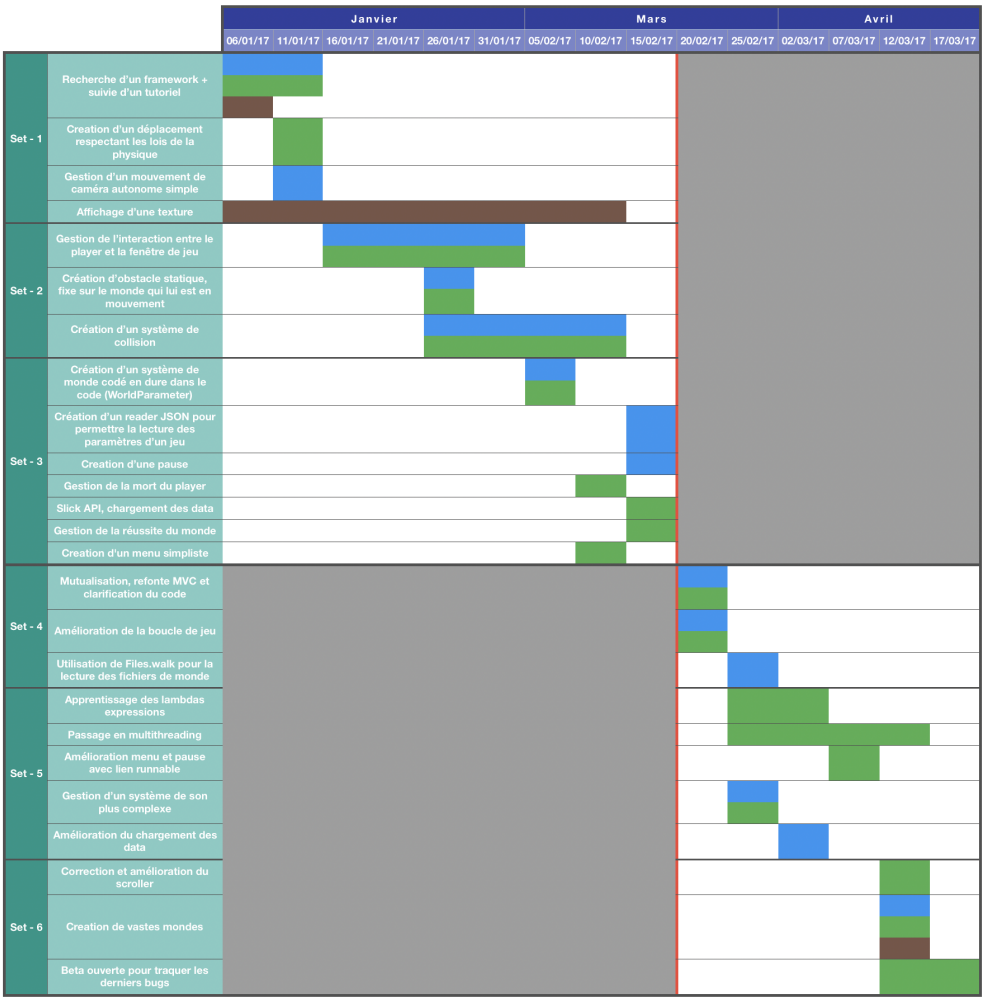

FIGURE 4 – diagram de gantt

3.1.4 Gestion des collisions

Après avoir implémenté des obstacles, nous avons décidé de gérer les collisions entre le joueur et ces derniers. Cette tâche a été le premier défi technique que nous avons rencontré.
Pour résoudre ce problème, nous avons implémenté des objets de type PotentialCollision. Ce sont des couples qui associe au joueur un obstacle du monde. À chaque update du jeu, chaque PotentialCollision est interrogé afin de savoir si le joueur est en contact avec un obstacle, et le cas échéant, sur quel bord de ce dernier la collision a lieu.
Ainsi, dans le cas où le joueur bloqué par un obstacle qui l'empêche d'avancer vers la droite, il est inutile de vérifier sur s'il est en contact avec le bord gauche d'un autre obstacle.

3.1.5 Création d'un parser JSON

Afin de pouvoir choisir entre plusieurs mondes au lancement du jeu, nous avons décidé de stocker les données des différents niveaux dans des fichiers JSON qui seraient lu au moment où l'utilisateur choisit le monde auquel il veut jouer.
Chaque fichier contient les toutes les informations nécessaires à l'instanciation du monde souhaité, telles que les textures à afficher, la musique à jouer, le placement des obstacles et de la sortie, le type de caméra, ou encore la puissance de la gravité qui s'applique au joueur.
Ce système nous a permis par la suite de créer des mondes variés et de s'affranchir des contraintes liées au fait de stocker les données des différents mondes directement dans le code du jeu.

4 Quelques défis techniques

4.1 Gestion des collisions

4.2 Le lambda calcul, multithreadé

Cette partie à était faite par Wissame, l'idée du lambda calcul est apparut au milieu du projet quand il a fallut gérer les bouton du menu, le fait d'associer un bouton à quelque chose que l'ont pouvait executer directement était séduisante pour la clarté du code mais aussi pour le défis technique que

cela représentais. Après quelque recherche la classe Runnable est apparût comme nécessaire pour la réalisation.
La classe Runnable est une classe qui existe depuis des années dans le langage JAVA mais depuis JAVA 1.8 il est possible de définir un Runnable à partir d'une lambda expression.
Un Runnable peut donc être définie par une lambda expression, cependant cette expression ne peut prendre aucune entrée et ne renvoie rien en sortie.

4.2.1 Le lambda calcule en JAVA

Depuis l'API JAVA 1.8, il est possible en JAVA de faire du lambda calcul, pour cela on dois passer des objets spécifique. Dans ce projet nous avons eu besoin des objets Runnable, Callable et Consumer.

Runnable Définie par un lambda calcul qui n'accepte aucun paramètre et qui ne renvoie rien.

Callable Définie par un lambda calcul qui prend renvoie quelque chose.

Consumer Définie par un lambda calcul qui prend quelque chose entrée mais qui ne renvoie rien

Il existe bien évidement d'autre type d'objet pour être définie par des lambda expressions plus complexe.

Nous avons par exemple utilisé les Runnable pour définir l'execution de l'action adéquat en cas de collision avec un objet. Mais aussi pour gérer de manière plus correcte la victoire et la mort du joueur.
Les Consumer ont servie pour le parcours de liste, avec la méthode foreach d'un objet Collection il est en effet possible d'effectuer une action sans avoir besoin d'un Iterator, en plus d'un gain de code, on limite le nombre d'erreur en supprimant les effets de bords.
Les Callable quant à eux ont était nécessaire pour la parallélisation des tâches.

4.2.2 ExecutorService et Future

Comme vue précédement les callables ont permis la parallélisation des tâches, associer à ExecutorService on peut en effet parallèliser simplement des tâches définie par des lambda expressions.
Il existe plusieurs manière d'utiliser un ExecutorService, dans ce projet je l'ai utiliser de deux manières avec un submit et avec un invokeAll.

La fonction submit d'un ExecutorService permet d'executer une action simple sur un autre thread, cette fonction renvoie un objet de type Future qui renseigne sur l'état d'avancement, on peut donc savoir quand est-ce qu'elle est finie, si on a besoin d'attendre avant de passer à la suite.
De plus, l'ExectureService permet d'executer une collection de Callable grace à la function invokeAll, cette function prend donc une collection de Callable et va l'executer sur le nombre maximale de thread sur lequel il à était définie. On a pas besoin de récupérer de future cette fonction est bloquante, on ne passe à la suite que si toute les Callable sont fini.

Table des figures