Rapport Final

du

Projet de L3

GAME FACTORY

Réalise par :

Simon ARNOULT
Wissame MEKHILEF
Vincent RENARD

Table des matières

1 Introduction

Le jeu que nous vous présentons dans ce court rapport est un jeu 2D en vue de côté. Le sujet imposé étant peu contraignant, nous avons essayé de varier les mouvements de caméra afin de proposer plusieurs jeux en un.
Le gameplay a été autant inspiré de jeux de plates-formes comme Super Mario Bros. que de jeux en side-scrolling comme Jetpack Joyride.

2 Présentation globale

2.1 Diagramme de classe

Le diagramme de classe visible en figure 1 donne un aperçu de l'état des classes et de leurs liens entre elles.
Il permet de se rendre compte du rôle prépondérant de la classe WorldReader, qui sera décrite ci-après.

2.2 Diagramme des boucles de jeu

2.2.1 Les contextes

Le contexte est un concept que nous avons utilisé pour déterminer l'état dans lequel se trouve le jeu. Il détermine quelles données doivent être mises à jour et quels éléments du monde doivent être affichés.
Les transitions d'un contexte à un autre sont décrites en figures 2 et 3.

2.2.2 La boucle update

La vitesse du jeu dépend de la fréquence des « ticks ». Chaque tick est un marqueur temporel séparé de son prédécesseur par une très courte durée fixée – dans notre programme, deux ticks consécutifs sont séparés par un soixantième de seconde.
La mise à jour des données du jeu s'effectue à chaque tick. Cela permet à notre programme de s'exécuter à une vitesse constante, et surtout indépendante de la puissance de la machine de l'utilisateur.

La figure 4 montre comment se déroule la mise à jour des données en cours de partie.

2.2.3 La boucle render

Contrairement à la mise à jour des données, l'actualisation de l'affichage n'a pas besoin d'être bridé : en effet, un nombre élevé de FPS (frames per second) permet d'obtenir un rendu à l'écran plus fluide.

2.3 La classe Graphics

Afin de séparer la vue des données, nous avons décidé de déléguer les méthodes d'affichage à une classe non instanciable nommée « Graphics » (cf. figure 5).

2.4 La classe Physics

De la même manière, nous avons placé le contrôleur – c'est-à-dire l'ensemble des méthodes influant sur les mouvements du joueur ainsi que ses interactions avec les éléments du monde – dans une classe nommée « Physics ».

3 Une organisation sans faille

Durant le projet, nous avons essayé d'avancer étape par étape afin de ne pas nous disperser et de garder un code cohérent.

3.1 Description des tâches

3.1.1 Recherche d'un framework

Nous avons choisi d'utiliser la librairie LWJGL [1] afin de gérer l'affichage graphique de notre jeu. Nous avons ensuite suivi le premier tiers du tutoriel

1. Lightweight Java Game Library en version 2.9.3

« 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, Simon a 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.

3.1.4 Gestion des collisions

Après avoir implémenté des obstacles, Wissame et Simon ont décidé de gérer les collisions entre le joueur et ces derniers. Cette tâche a été le premier défi technique que nous avons rencontré.

3.1.5 Création d'un parser JSON

Afin de pouvoir choisir entre plusieurs mondes au lancement du jeu, Simon a 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.

3.1.6 Ajout des textures, du son et des polices d'écriture

Après que Vincent a essayé pendant plusieurs semaines d'implémenter des textures afin que le jeu puisse rendre à l'écran d'autres éléments que

des tuiles monochromes, Wissame a décidé d'utiliser la librairie Slick afin d'enrichir l'affichage du jeu.
En plus de lui permettre de résoudre rapidement le problème, Slick nous a permis par la suite d'ajouter un fond sonore à notre jeu, ainsi que des polices d'affichage nécessaires à la création d'un menu.

4 Quelques défis techniques

4.1 Gestion des collisions

Pour résoudre ce problème, Wissame et Simon ont 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 est 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.

4.2 Le lambda calcul en multithread

L'idée du lambda calcul est apparue lorsqu'il nous a fallu gérer les boutons du menu. Le fait d'associer à chaque bouton une action exécutable directement était séduisante pour la clarté du code mais aussi pour le défi technique que cela représentait. Après quelques recherches, Wissame se dirigea vers la classe Runnable.
La classe Runnable est une classe qui existe depuis plusieurs années dans le langage Java, or depuis Java 1.8 il est possible de définir un Runnable à partir d'une lambda expression. Cependant, cette expression ne peut prendre aucun argument en entrée et ne retourne rien.

4.2.1 Le lambda calcul en Java

Depuis Java 1.8, il est possible en de faire du lambda calcul,en passant par des classes spécifiques. Dans ce projet, nous avons eu besoin des objets Runnable, Consumer et Callable.

Runnable Défini par une lambda expression qui n'accepte aucun paramètre et ne renvoie rien.

Consumer Défini par une lambda expression qui peut avoir des arguments en entrée mais qui ne renvoie rien.

Callable Défini par une lambda expression qui peut avoir des arguments en entrée et avoir un type de retour non vide.

Il existe évidemment d'autres types d'objets définis par des lambda expressions plus complexes.
Nous avons par exemple utilisé les Runnable pour définir l'execution de l'action adéquate en cas de collision avec un élément du monde, ainsi que pour gérer de manière plus correcte la victoire ou la mort du joueur.
Les Consumer ont servi dans le cas d'un parcours de listes ; en effet, il est possible en parcourant un objet Collection d'effectuer une action sans avoir besoin d'un Iterator, ce qui, en plus d'un gain de code, limite le risque d'erreur en supprimant les effets de bords.
Les Callable, quant à eux, ont été nécessaires pour la parallélisation des tâches.

4.2.2 ExecutorService et Future

Associés à un ExecutorService, les Callable permettent de paralléliser simplement des actions définies par des lambda expressions.
Il existe plusieurs manières d'utiliser un ExecutorService. Wissame l'a utilisé de deux manières : avec un submit et avec un invokeAll.
La fonction submit d'un ExecutorService permet d'exécuter une action simple sur un thread. Cette fonction renvoie un objet de type Future qui indique l'état d'avancement de l'action.
De plus, l'ExecutorService permet d'exécuter une collection de Callable grâce à la fonction invokeAll. Ainsi, les actions sont exécutées sur le plus grand nombre de threads possible. De cette manière, il est inutile de récupérer un Future, puisque la suite du programme ne s'effectue que si tous les Callable ont été exécutés.

Table des figures

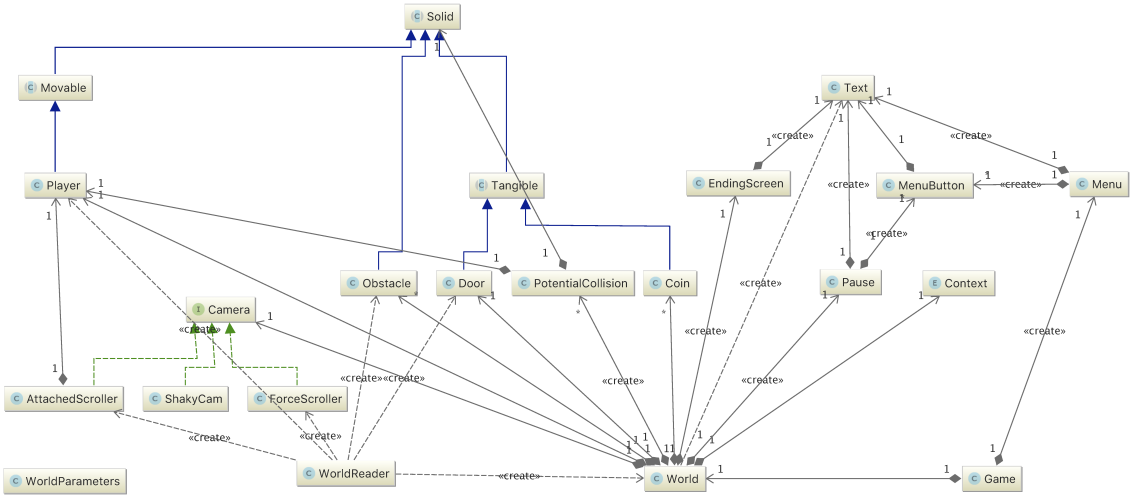

FIGURE 1 – Le diagramme de classe UML

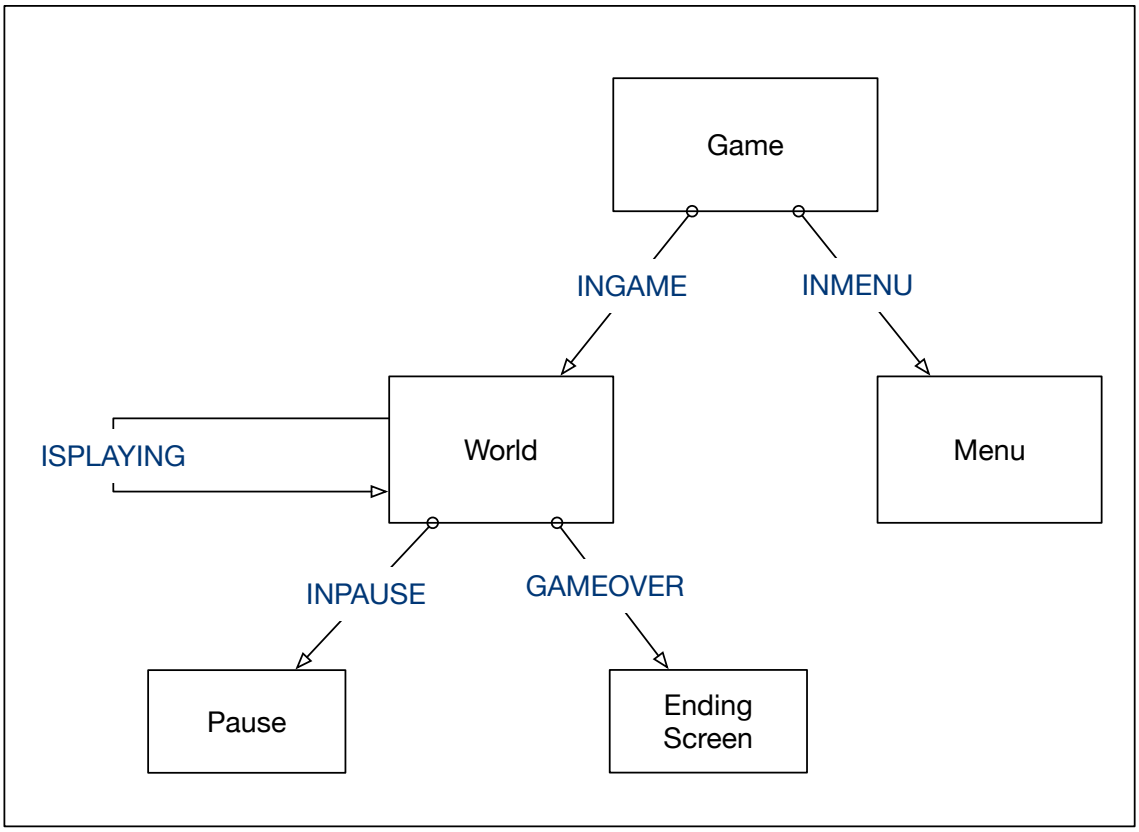

FIGURE 2 – schema 1

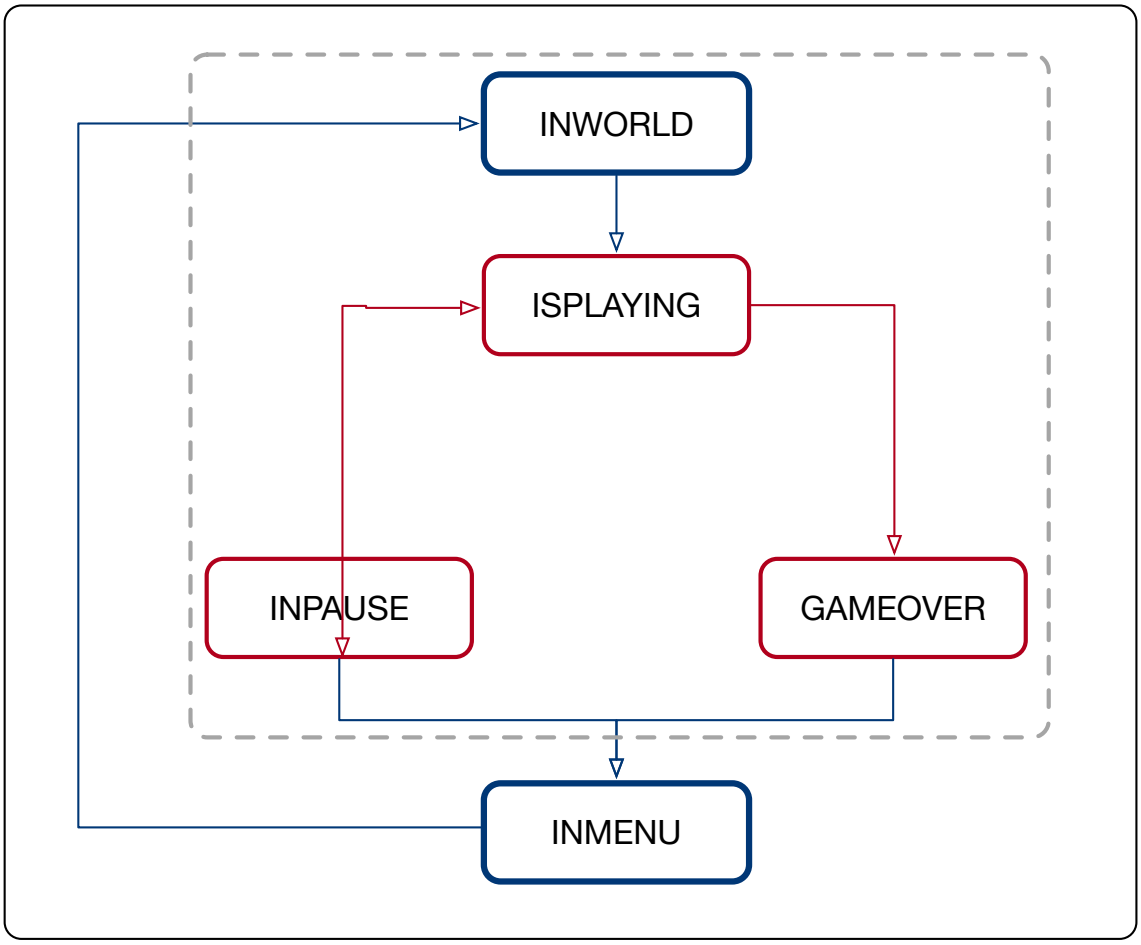

FIGURE 3 – schema 2

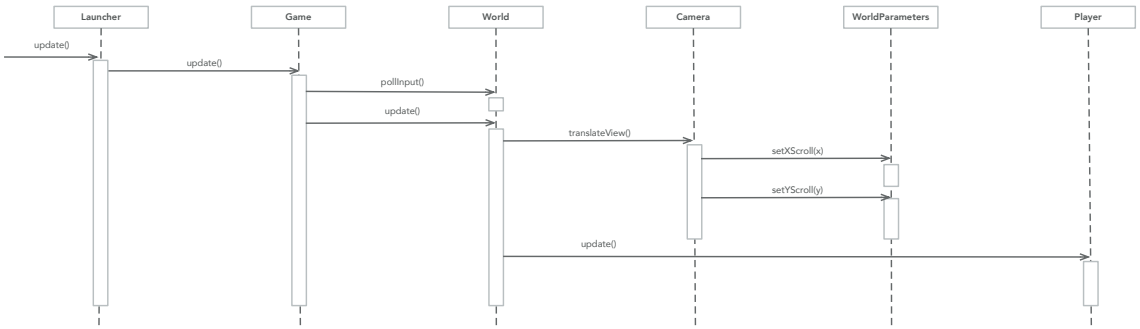

FIGURE 4 – Diagramme Séquence Système de la fonction Update

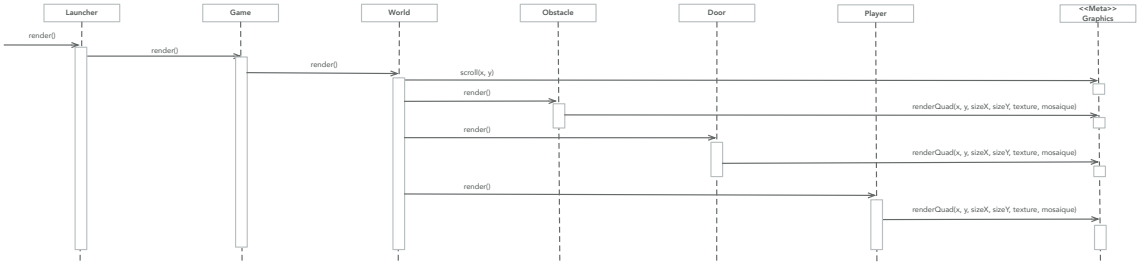

FIGURE 5 – Diagramme Séquence Système de la fonction Render

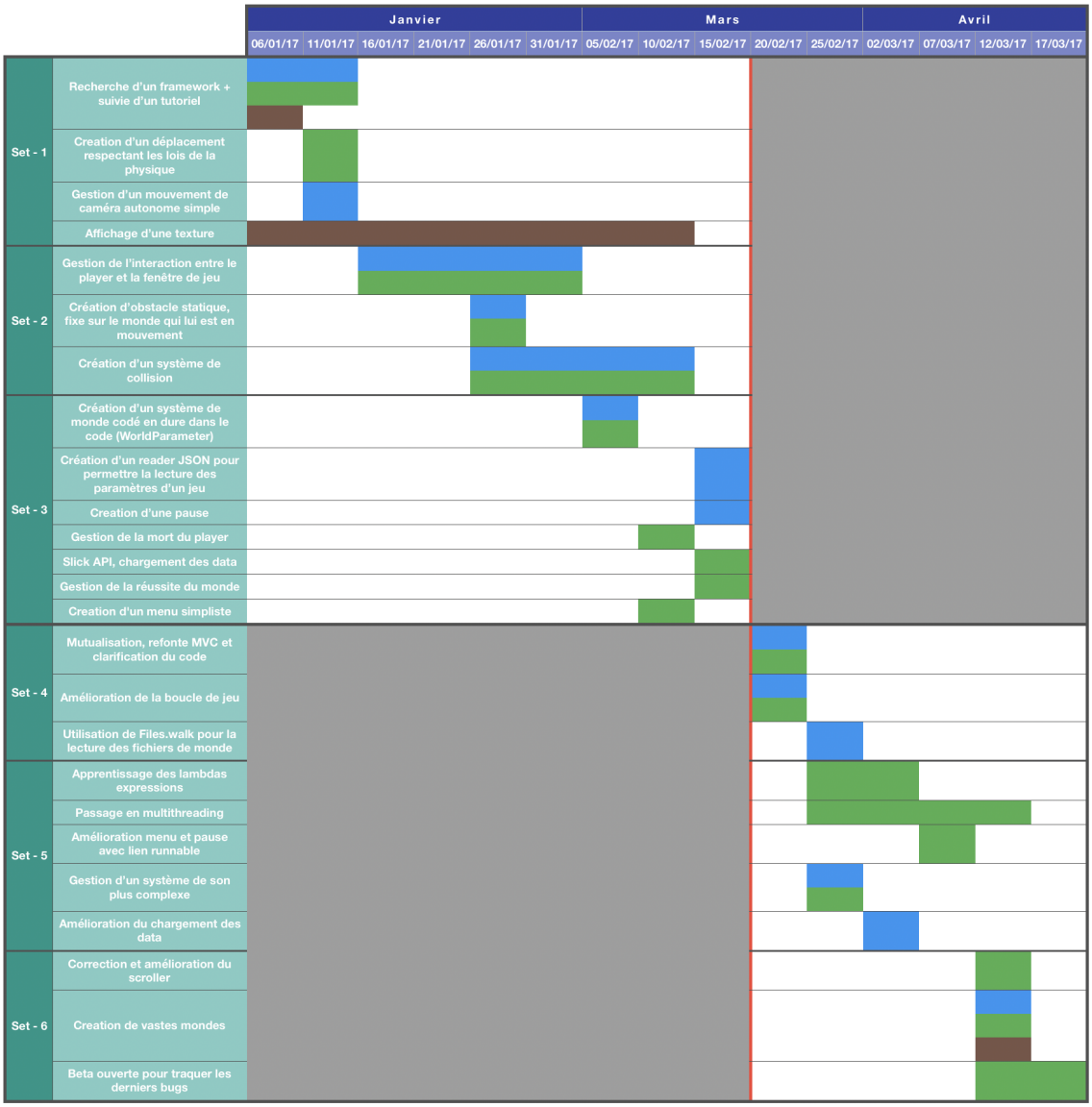

FIGURE 6 – diagram de gantt