



Rapport Final du **Projet de L3**

GAME FACTORY

Réalise par :

Simon ARNOULT
Wissame MEKHILEF
Vincent RENARD

Table des matières

1	Introduction	3
2	Présentation globale	3
2.1	Diagramme de classe	3
2.2	Diagramme des boucles de jeu	4
2.2.1	Les contextes	4
2.2.2	La boucle update	4
2.2.3	La boucle render	6
2.3	Architecture MVC	6
2.3.1	Affichage des données	6
2.3.2	Interpretation des entrées clavier	6
3	Une organisation sans faille	6
3.1	Description des taches	6
3.1.1	Recherche d'un framework + suivi d'un tutoriel	6
3.1.2	Gestion d'un mouvement de caméra simple et autonome	6
3.1.3	Gestion de l'interaction entre le joueur et la fenêtre . .	8
3.1.4	Gestion des collisions	8
3.1.5	Création d'un parser JSON	8
4	Quelques défis techniques	9
4.1	Gestion des collisions	9
4.2	Le lambda calcul, multithreadé	9

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.

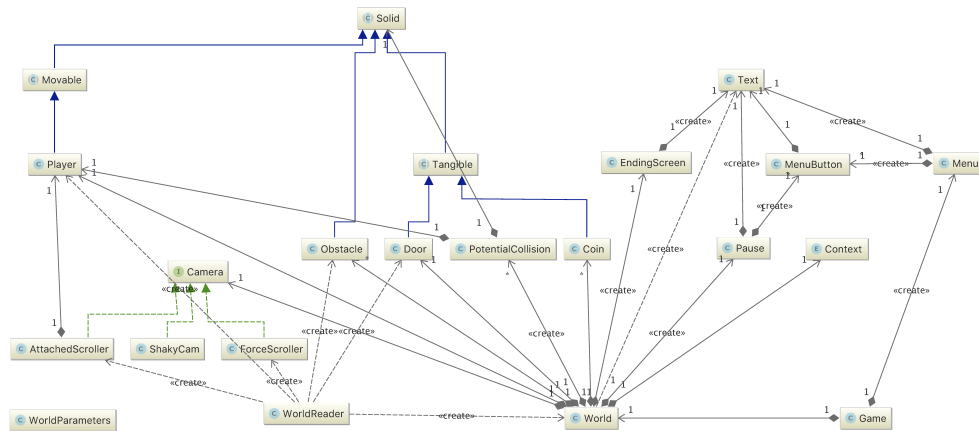


FIGURE 1 – Le diagramme de classe UML

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 rendre que des éléments nécessaire.

De plus il nous ont permis de définir des règles pour passer d'un contexte a un autre.

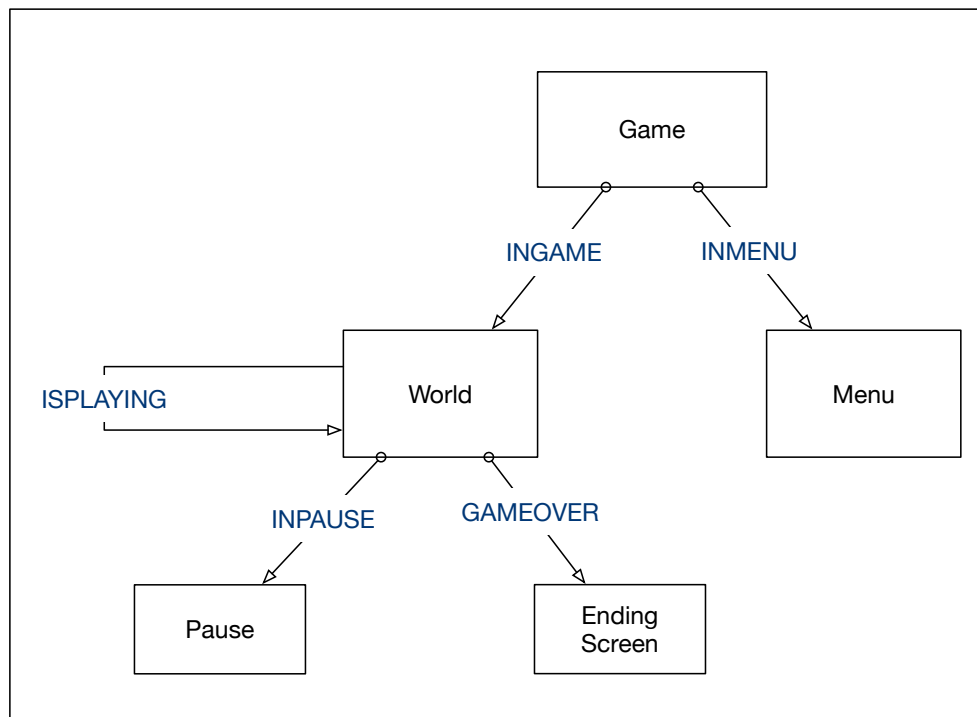


FIGURE 2 – schema 1

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

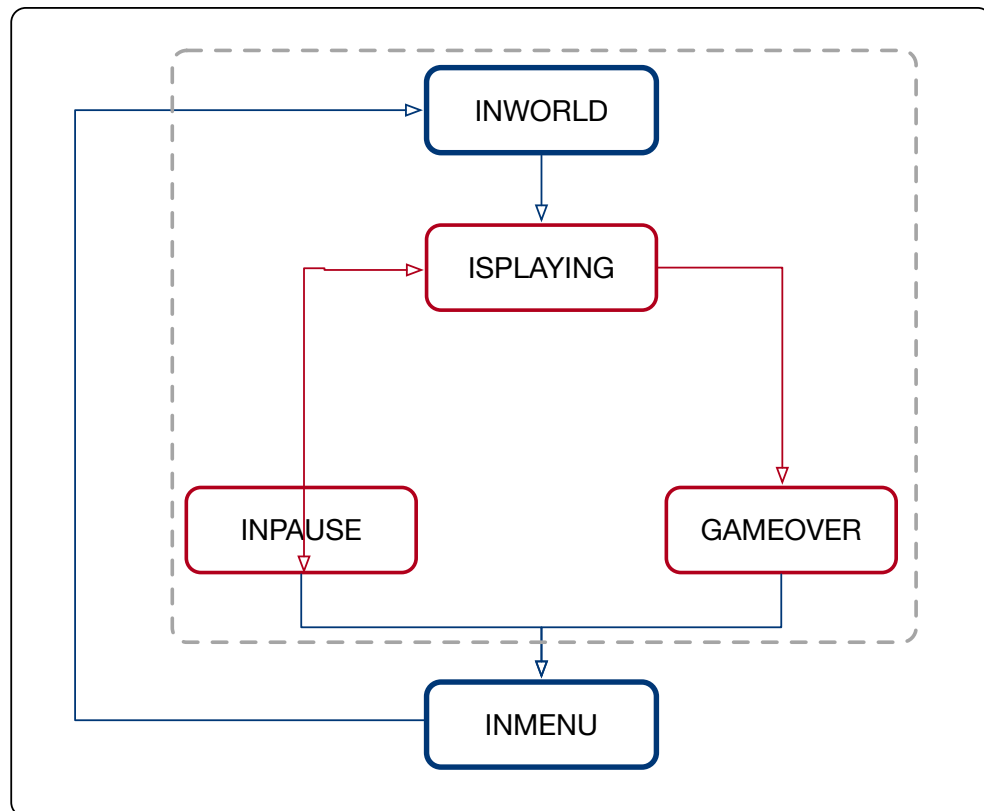


FIGURE 3 – schema 2

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.

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 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.

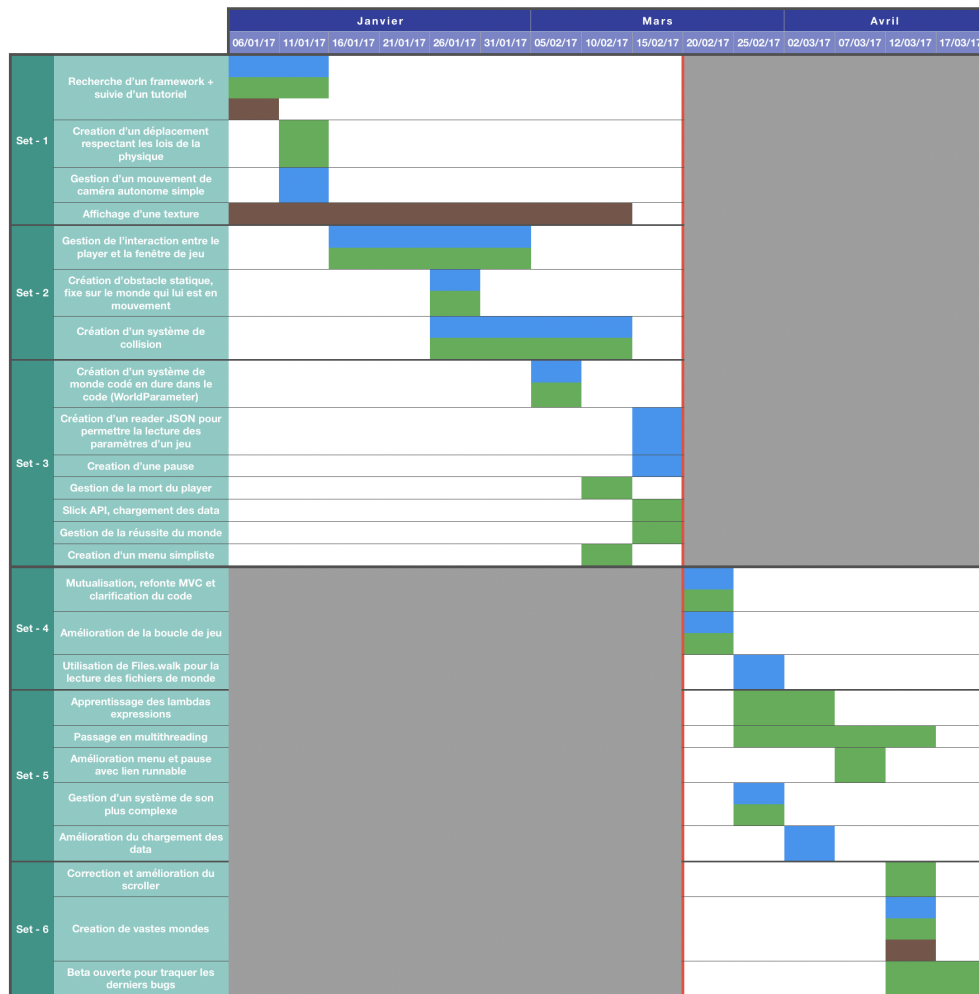


FIGURE 4 – diagram de gantt

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, 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 apparu 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. Elle est donc adaptée si c'est une suite de fonction qui peut être exécutés indépendamment des autres. Rapidement il a fallut trouver d'autres types d'objet pour faire définir un objet q

Table des figures

1	Le diagramme de classe UML	3
2	schema 1	4
3	schema 2	5
4	diagram de gantt	7