



Rapport Final
du
Projet de L3

GAME FACTORY

Réalisé 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	3
2.2.1	Les contextes	3
2.2.2	La boucle update	3
2.2.3	La boucle render	4
2.3	La classe Graphics	4
2.4	La classe Physics	4
3	Une organisation	4
3.1	Description des taches	5
3.1.1	Recherche d'un framework	5
3.1.2	Création d'un déplacement respectant les lois de la physique	5
3.1.3	Gestion d'un mouvement de caméra simple et autonome	5
3.1.4	Gestion de l'interaction entre le joueur et la fenêtre . .	5
3.1.5	Gestion des collisions	5
3.1.6	Création d'un parser JSON	5
3.1.7	Ajout des textures, du son et des polices d'écriture . .	6
4	Quelques défis techniques	6
4.1	Gestion des collisions	6
4.2	Le lambda calcul, multithreadé	7
4.2.1	Le lambda calcul en JAVA	7
4.2.2	ExecutorService et Future	8
5	Conclusion	8

1 Introduction

Sans avoir à rappeler le sujet, nous avons choisit de programmer un monde 2D avec une vue latérale, les mouvements de caméra au début n'étaient pas clairement définis.

Quant au gameplay, on voulait un jeu simple à jouer avec une idée empruntée au jeu de runner où seul le saut est possible.

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 par exemple que WorldReader cree tout ce qui est nécessaire pour le monde.

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.

À chaque contexte est associé un objet principal, ces associations sont décrite sur la figure 2.

Les transitions d'un contexte à un autre sont décrites sur la figure 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

On a toujours durant le projet avancer par étapes nécessaire pour ne pas se perdre. La répartition des tâches est détaillée dans le diagramme de GANTT (cf. figure 6)

3.1 Description des taches

3.1.1 Recherche d'un framework

Nous avons choisi d'utiliser la librairie LWJGL¹ afin de gérer l'affichage graphique de notre jeu. Nous avons ensuite suivi le premier tiers du tutoriel "Jeu 2D avec LWJGL" sur la chaine YouTube "Tuto Programmation (Marccspro)" afin de nous familiariser avec cette librairie.

3.1.2 Création d'un déplacement respectant les lois de la physique

Cette partie a été réalisée par Wissame, dès que le joueur fut affiché à l'écran, il a voulu qu'il puisse se déplacer de manière naturelle. Pour cela, il a utilisé Matlab pour tester les fonctions de gravité avant de les implémenter dans le jeu.

3.1.3 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.

3.1.4 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.

3.1.5 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.

3.1.6 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

1. Lightweight Java Game Library en version 2.9.3

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.

3.1.7 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 de résoudre ce problème bloquant, la librairie Slick a permis de récupérer des objets texture qui ont pu directement être utilisés dans la fonction de rendu. En plus de lui permettre de résoudre rapidement le problème, Slick lui a permis par la suite d'ajouter un fond sonore au jeu, ainsi que de rendre du texte avec une police particulière, de la même manière que pour les textures, des objets Font et Sound sont disponibles grâce à cette API.

Slick étant conçue comme une extension de LWJGL l'ajout toutes ces fonctions n'a pas posé de problèmes.

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 eu lieu.

De plus cette séparation des traitements obstacle par obstacle permet des les traiter dans un ordre quelconque, en effet si un PotentialCollision détecte une collision sur un coté de l'obstacle le Player reçoit cette information donc toutes les autres PotentialCollision reçoivent l'information.

Donc si un `PotentialCollision` détecte une collision sur le haut de l'obstacle alors le `Player` reçoit l'information comme quoi il est bloqué par le bas et comme le `Player` reste un objet unique qui n'est pas dupliqué, toutes les autres `PotentialCollision` prennent en compte cette information. C'est grâce à cela que le calcul des collisions a pu être parallélisé, la variable `isStuckRoutine` contenant l'appel à tout les calculs de collision. Nous expliquons dans la partie qui suit grâce à quoi cela fonctionne.

4.2 Le lambda calcul, multithreadé

Cette partie a été faite par Wissame, l'idée du lambda calcul est apparue au milieu du projet quand il a fallut gérer les boutons du menu, le fait d'associer un bouton à quelque chose que l'on pouvait exécuter directement était séduisante pour la clarté du code mais aussi pour le défis technique que cela suscitait. Après quelques recherches, la classe `Runnable` est apparue 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'un lambda calcul.

Un `Runnable` peut donc être défini par un lambda calcul, cependant ce calcul ne peut prendre aucune entrée et ne renvoie rien en sortie.

4.2.1 Le lambda calcul en JAVA

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

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

Callable Défini par un lambda calcul qui ne prend rien et renvoie quelque chose.

Consumer Défini par un lambda calcul qui prend quelque chose en entrée mais qui ne renvoie rien.

Il existe bien évidemment d'autres types d'objets pour être définis par des lambda expressions plus complexes.

Nous avons par exemple utilisé les `Runnable`s pour définir l'exécution de l'action adéquat en cas de collision avec un objet. Mais aussi pour gérer de manière plus générale la victoire et la mort du joueur.

Les `Consumers` ont servis pour le parcours de liste, avec la méthode `foreach` d'un objet `Collection`. Il est en effet possible d'effectuer une action pour chacun des objets de la liste sans avoir besoin d'un `Iterator`, en plus d'un gain de code, on limite le nombre d'erreurs en supprimant les effets de bords. Les `Callables` quant à eux ont été nécessaires pour la parallélisation des tâches.

4.2.2 `ExecutorService` et `Future`

Comme vue précédemment les `Callables` ont permis la parallélisation des tâches. Associés à un `ExecutorService` on peut en effet paralléliser simplement des tâches définies par des `lambda` expressions.

Il existe plusieurs manière d'utiliser un `ExecutorService`, dans ce projet je l'ai utilisé de deux manières, avec un `submit` et avec un `invokeAll`.

La fonction `submit` d'un `ExecutorService` permet d'exécuter une action sur un autre thread, cette fonction renvoie un objet de type `Future` qui renseigne sur l'état d'avancement de la tâche, on peut donc savoir quand est-ce qu'elle est finie si on a besoin d'attendre la fin avant de passer à la suite.

De plus, l'`ExecutorService` permet d'exécuter une collection de `Callable` grace à la fonction `invokeAll`, cette fonction prend donc une `Collection` de `Callable` et va l'exécuter sur le nombre maximal de `Thread` sur lequel l'`ExecutorService` a été définie. On n'a pas besoin de récupérer de `Future`. Cette fonction est bloquante, on ne passe à la suite que si toutes les exécutions des `Callables` sont finies.

5 Conclusion

Ce projet est maintenant fini et comme pour tout projet il y'a des objectifs qui ont été atteints et d'autres non.

Pour ce qui est des objectifs fixés, nous avons un jeu qui fonctionne. Le code est généralisé ce qui permettrait d'ajouter des nouveaux types d'objets facilement et de décrire l'interaction avec ces derniers. Les différents types de caméra sont eux aussi pleinement fonctionnels ce qui permet de proposer des gameplay différents. Les règles de la physique du monde sont elles aussi

modifiables ce qui permet d'améliorer l'expérience de jeu. Pour finir le calcul parallèle permet de profiter des architectures multi-cœur.

En ce qui concerne l'ajout de nouvelles fonctions, car il y'en aura toujours. On pourrait améliorer la gestion des collisions, car il y'a encore un bug avec deux obstacles contigus. On aurait pu jouer aussi sur le bruitage avec l'ajout de son lors d'une collision. On pourrait ajouter aussi de nouveaux types d'objets dans le monde tel que des monstres ou des pièces, tout le système de gestion étant déjà présent.

Voici le lien vers le dépôt Github du projet : https://github.com/wissame95/Game_Factory

Table des figures

1	Le diagramme de classe UML	11
2	Schéma de l'association contexte et objet principal associé . .	12
3	Schéma des transitions entre contexte	13
4	Diagramme Séquence Système de la fonction Update	14
5	Diagramme Séquence Système de la fonction Render	14
6	Diagramme de GANTT	15

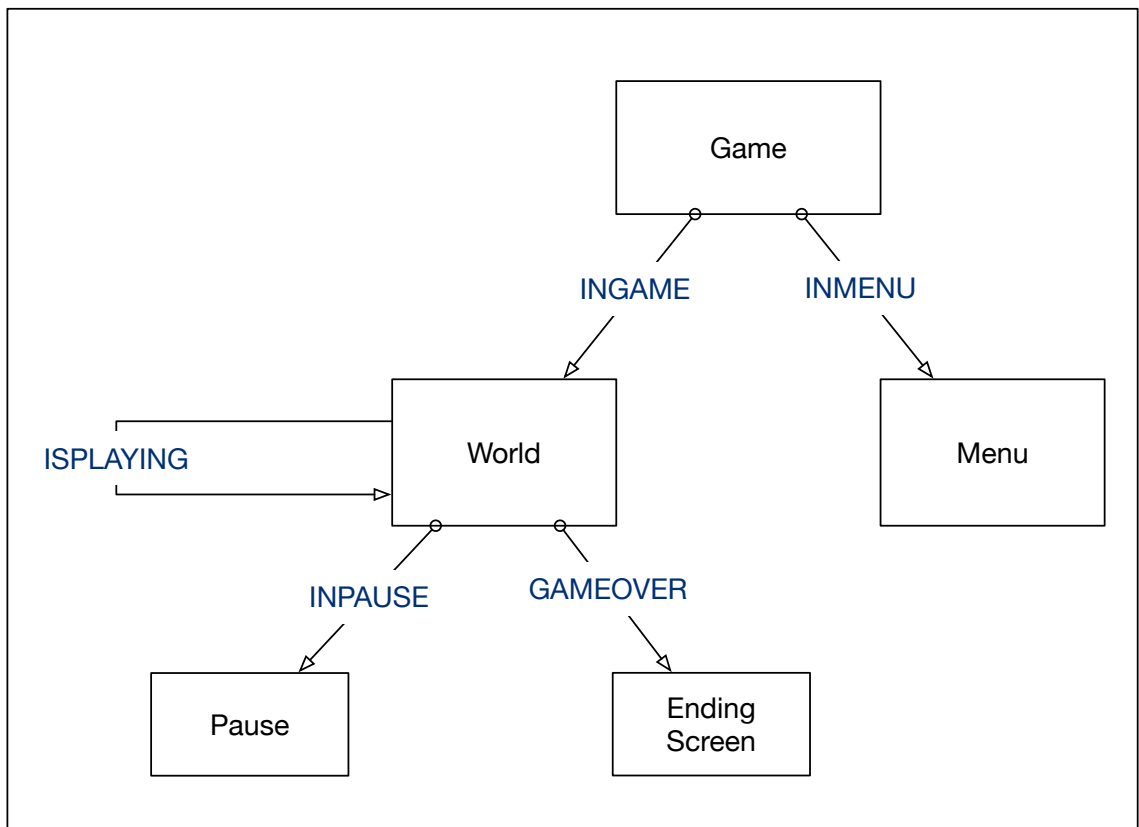


FIGURE 2 – Schéma de l'association contexte et objet principal associé

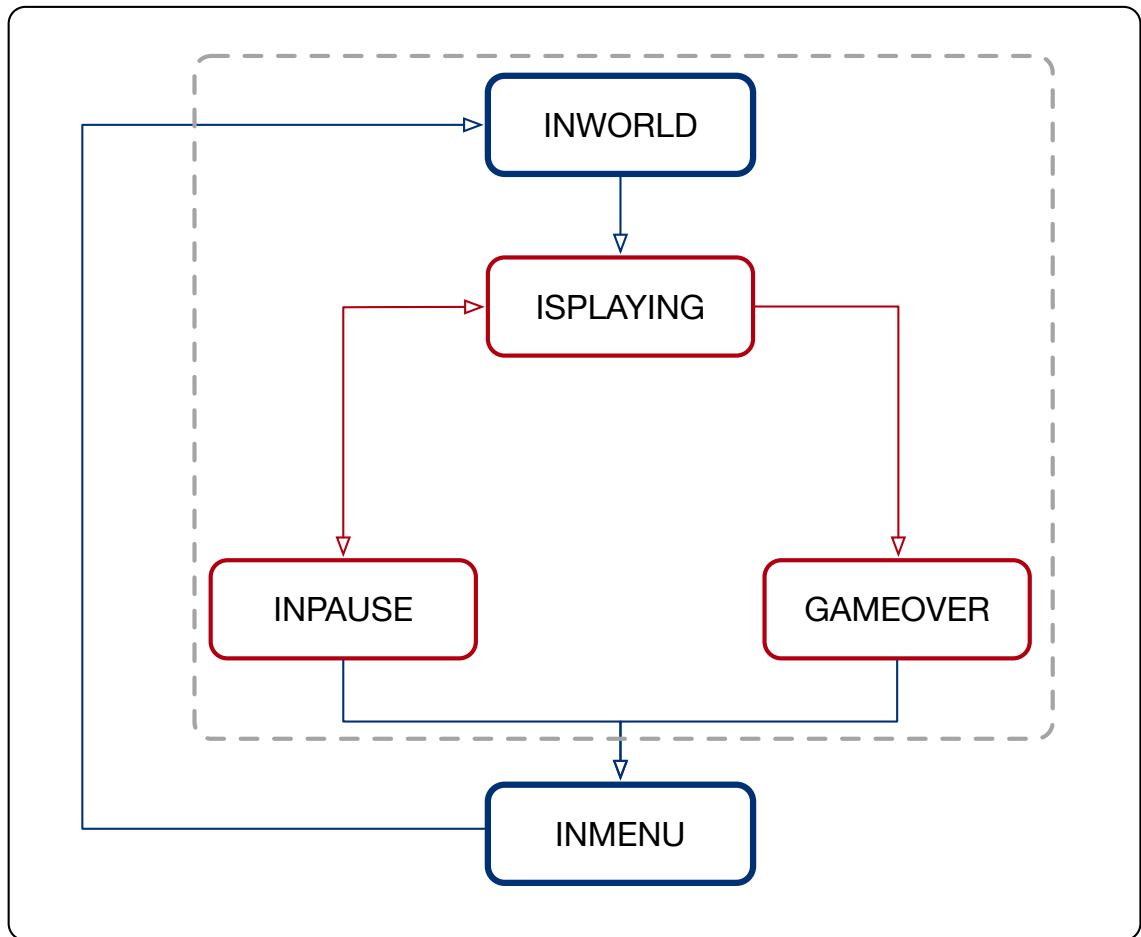


FIGURE 3 – Schéma des transitions entre contexte

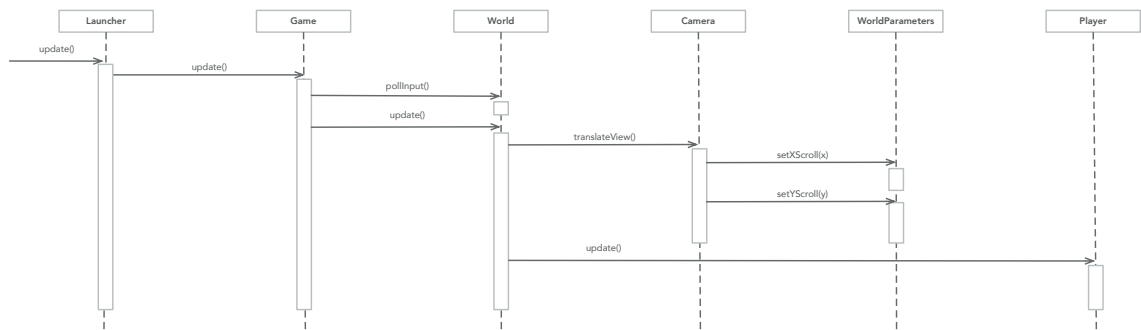


FIGURE 4 – Diagramme Séquence Système de la fonction Update

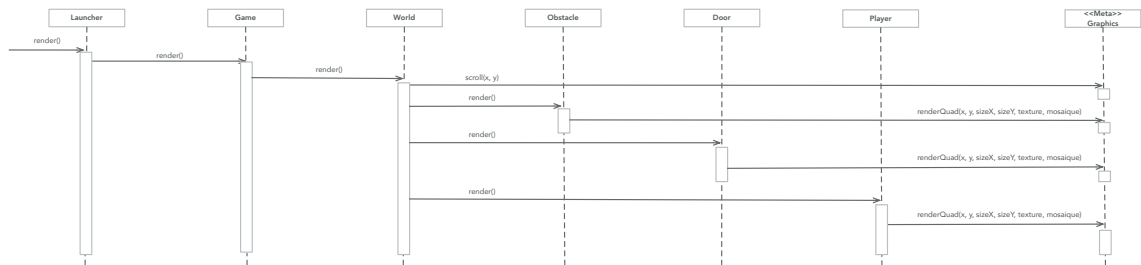


FIGURE 5 – Diagramme Séquence Système de la fonction Render

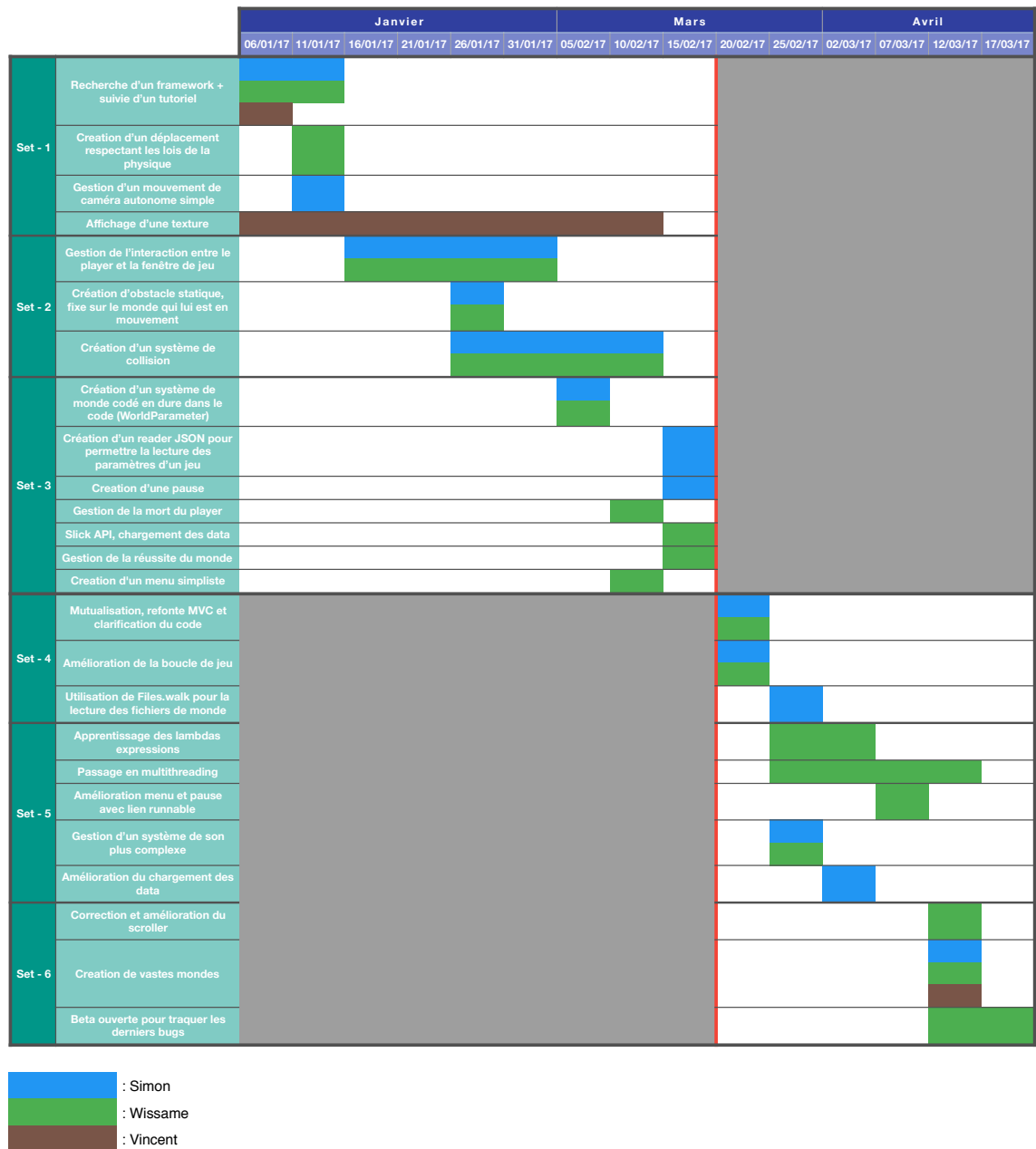


FIGURE 6 – Diagramme de GANTT