
Université d'Orléans
Collegium Sciences et Techniques
Département Informatique
Licence 3 - Ingenierie Informatique
Semestre 6

# Compte Rendu Projet

Maria Lost - Projet Programmation S6

Collet Elsa - 21 55 411
Crédeville Louis-Maxime - 21 30 077
Labourbe Loic - 21 36 709

# Sommaire

# 1 Introduction

L'objectif de ce projet est de développer un jeu. Un personnage est perdu dans un labyrinthe et cherche à rejoindre la sortie en ramassant le plus d'or possible et en tuant les ennemies. Ce jeu est développé en Java8 et utilise Javafx. Nous avons étroitement collaboré pour obtenir une version bêta puis nous avons implémenté d'autres parties avec plus autonomie. Le code est organisé en différents packages *MainApp* pour lancer l'application, *Item* qui regroupe les objets, *User* principalement utilisé par le menu et *Play* qui gère le plateau.

# 2 Présentation

## 2.1 Fonctionnalités

L'objectif de ce jeu est de faire sortir Maria des différents labyrinthes en faisant le plus gros score possible. Pour cela, le joueur doit trouver la sortie en ramassant l'or qu'il trouve, et tuer les ennemis qui se transforment alors en or.
Au départ, le jeu est très simple, ce qui permet de se familiariser avec les commandes. En augmentant le niveau du personnage, des ennemis commencent à apparaître ; plus le niveau augmente, plus leur nombre grandit.
Le nombre de niveaux est infini. Á chaque niveau obtenu, la taille du labyrinthe augmente jusqu'à une certaine limite.

## 2.2 Choix technologiques

Nous avons décidé de coder l'application avec Java8 et d'utiliser la bibliothèque Javafx. Nous avons utilisé GIT comme outil de versionnage et des logiciels comme SceneBuilder pour la création de fichiers .fxml en WYSIWYG et ScenicView pour déboguer la vue.

## 2.3 Organisation du travail

Nous avons dans un premier temps expérimenté seuls les technologies retenues pour nous former. Loic s'est principalement intéressé à la représentation et l'affichage des images, Louis-Maxime au mouvement du personnage et Elsa à la fenêtre d'accueil et l'organisation MVC de l'application.
Elsa a ensuite regroupé les différentes parties pour créer une première version très simple. Nous avons ensuite collaboré pour régler les différents problèmes qui se présentaient d'un point de vue architectural et dans la gestion d'élément instable. Louis-Maxime et Elsa ont réarrangé beaucoup de code. Louis-Maxime s'est penché sur les animations et le fonctionnement de *World*.
Une fois la version stable terminée, Loic s'est chargé de travailler sur la partie ennemie et Elsa sur la génération le labyrinthe ainsi que la fin de partie.

**Cf Annexe 1 : Diagramme de Gantt**

# 3 Organisation du code

**Cf Annexe 2 : Diagramme UML**

## 3.1 *Parameters*

Nous avons décidé de sortir un grand nombre d'informations en information static dans le fihier *Parameters_MariaLost*. Cette méthode nous a permis de manipuler plus facilement l'application et nous rendre plus indépendants les uns des autres. Nous envisagions de créer un fichier de propriété pour générer cette classe de paramètre et laisser la possibilité à l'utilisateur de modifier ces informations. Nous n'avons cependant pas poursuivi cette possibilité.

## 3.2 *MainApp*

*MainApp* permet de lancer l'application. Elle crée le *primaryStage* et l'unique *Starter*. *Starter* correspondrait à un contrôleur frontal. Les autres contrôleurs possèdent une instance de celle-ci. Il permet de charger la liste des joueurs, enregistré dans un fichier *.xml*, de travailler dessus, de lancer une vue et son contrôleur. Soit elle appelle des contrôleurs dans le *package user* pour la partie menue, soit elle appelle le contrôleur de *Play*. Elle retient aussi un *currentUser* pour facilité l'accès aux informations.

## 3.3 *User*

Dans un premier temps, l'application charge les joueurs en utilisant une classe *Wrapper* utilisée par la classe *Reader* pour lire et écrire dans le fichier *.xml*. Les fichiers de la vue *.fxml* sont édités avec SceneBuilder. Les contrôleurs permettent de spécifier des valeurs aux objets fxml et de définir des évènements.

## 3.4 *Items*

Nous considérons le plateau comme une succession d'objets. Nous avons défini des interfaces pour, regrouper ceux qui peuvent être activés, dessinés, ou encore être mobiles. Nous avons défini par dessus une interface *Item* permettant de définir des caractéristiques communes comme la position
Il existe ensuite 3 classes abstraites :

- Abstract Item
- Abstract Mobil Item qui permet de gérer les déplacements
- Astract Enemy qui gère entre autres, les attaques.

### 3.4.1 Monnayeur et monnaie

Le joueur possède un monnayeur. Lorsqu'il rentre en contact avec un objet Money, il est crédité. Les autres objets possèdent un DebitOnly, ce qui les empêche de récupérer l'argent du plateau.

### 3.4.2 Animations

Les animations sont des tableaux de couple (durée,Image). À chaque appel pour afficher l'animation, on regarde le temps depuis la dernière demande et on récupère l'image correspondante.
Pour les ennemis, les images des animations sont stockées dans des *Srite Sheet*. Ceux-ci sont utilisés par la classe abstraite *SpriteSheetLoader*. Cette classe permet d'extraire une liste d'images d'un *Sprite Sheet* dans le but d'en faire une animation.

### 3.4.3 Ennemies

Les ennemis et leur comportement sont encapsulés dans la classe *AbstractEnemy*. Chaque ennemi possède une façon de se mouvoir et d'attaquer dans les quatre directions. Les animations sont encapsulées dans les instances des classes *MeleeAttack* et *Movement*.

**Comportement** Si l'ennemi est assez proche du joueur, il va choisir une attaque selon la direction du joueur. Si le joueur est éloigné, il va choisir le mouvement approprié pour se déplacer dans la direction du joueur. Si la position de l'ennemi n'a pas changé entre deux mouvements, c'est qu'il a rencontré un mur et qu'il est bloqué, on applique alors une sélection de mouvement différente pour essayer de se rapprocher du joueur.
Les mouvements et les attaques ont une durée finie, cela implique que tant que l'attaque ou le mouvement en cours n'est pas terminé, l'ennemi n'en choisit pas un nouveau.

**Les attaques à distances.** La classe *RangedAttack* permet de créer des projectiles tels que les boules de feu envoyées par le joueur. La classe *RangedAttack* hérite de la classe *AbstractItem*. De ce fait, on peut ajouter une instance de *RangedAttack* à la liste des items du monde, le moteur physique se chargera ensuite de la faire avancer.
Les araignées et squelettes héritent de la classe *AbstractEnemy*, seul leur constructeur est différent, mais on pourrait très bien redéfinir la méthode qui choisit les attaques ou les mouvements, et ainsi créer un ennemi avec un comportement totalement différent.

## 3.5 *GamePlay*

### 3.5.1 Contrôle du monde

Le principal contrôleur, *GameLayoutController*, permet de lancer le monde et de lui associer une vue. Les trois autres permettent de contrôler avec fxml les bannières d'informations et la page de fin de partie.
L'interface *Followable* permet de centrer le monde sur le personnage. L'interface *Model* permet de lancer le jeu.
La classe *World* imlémente *Modele*. Elle permet de regrouper un plateau, avec un personnage principal et la liste des objets posés sur le plateau ainsi que le moteur du jeu. Elle gère donc la boucle du jeu, en faisant bouger les *Items* et attaquer les ennemies. Elle retire aussi les *Items* qui ont terminé leur vie comme les *Money* qui n'ont plus d'argent, et les ennemies sans vie. Elle vérifie également si le joueur est mort ou s'il est arrivé à la fin du niveau.
Le moteur de jeu *PhysicMotor* gère les déplacements et les collisions.

### 3.5.2 Création de labyrinthes

On dispose de deux interfaces pour le plateau. *Floor* et *DrawableFloor* permettant de spécifier les éléments nécessaires à la création d'un plateau. On créer un *AbstractFloor* pour généraliser les méthodes entre les deux façons de créer un labyrinthe. Cette classe abstraite permet de récupérer les items dans un carré donné.
Les méthodes pour la création d'un labyrinthe enregistrent dans le tableau d'*Items* les objets fixes tels que le sol et les murs. L'or et les ennemis sont enregistrés dans une liste. *FloorFromFile* crée ces éléments à partir d'un fichier faisant correspondre un code (entier) à chaque objet. *GenerateLaby* crée un labyrinthe aléatoirement en suivant certaines règles. Pour ce dernier, on s'assure qu'il existe bien un chemin avant de valider le labyrinthe.

### 3.5.3 La vue

***FloorView*** calcule la portion du labyrinthe à afficher en fonction d'un point central, excepté au bord du labyrinthe. La mise à l'échelle permet d'obtenir une taille d'affichage indépendante de la taille de la portion du labyrinthe à afficher.
L'affichage à proprement parler se passe dans la méthode *draw*, elle prend une collection de *Drawable* qui ont une position, une taille et une image. L'affichage se fait en redimensionnant les images à la taille du *Drawable* et en la collant à sa position dans le repère de la fenêtre.

***GameView*** se charge de rafraichir périodiquement *FloorView* et d'associer à la vue les différentes bannières (concernant le joueur et celle concernant le personnage).
*GameView* récupère les évènements du clavier et de la souris :

- Pour le clavier : elle enregistre si une touche est activée et permet au *GameLayoutController* de récupérer l'état des touches pour effectuer l'action associée.
- Pour la souris : elle convertit la position du clic dans la fenêtre en position dans le repère du labyrinthe et transmet l'évènement, ce qui permet à *GameLayoutController* de lancer le déplacement du joueur et de lancer des boules de feu.

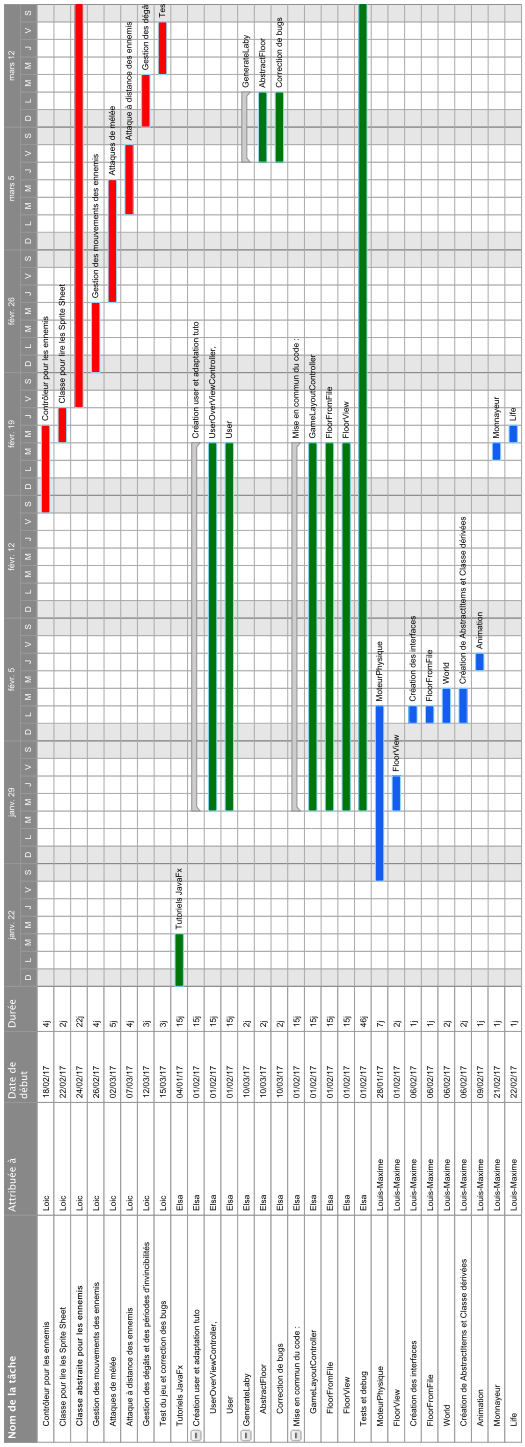
# 4 Conclusion

Certains dysfonctionnements se sont manifestés de manière très occasionnelle. Ils étaient difficiles à reproduire et à corriger. D'autres sont apparus lors de l'ajout de fonctionnalités. Ce qui nous a obligés à apporter de légère correction à du code qui fonctionnait correctement.
Nous connaissons maintenant des design patterns que nous aurions pu utiliser et qui nous aurais aidé dans la conception de l'application.
Nous avons apprécié le travail d'équipe fourni au cours de ce projet et nous pensons avoir globalement atteint nos objectifs et respecté la consigne d'une architecture Model Vue Contrôleur.

# 5 Annexes

## 5.1 Figure 1 - Diagramme de Gantt

Diagramme de Gantt



| Nom de la tâche | Attribuée à | Date de début | Durée |
|---|---|---|---|
| Contrôleur pour les ennemis | Loïc | 18/02/17 | 4j |
| Classe pour lire les Sprite Sheet | Loïc | 22/02/17 | 2j |
| Classe abstraite pour les ennemis | Loïc | 24/02/17 | 22j |
| Gestion des mouvements des ennemis | Loïc | 28/02/17 | 4j |
| Attaques de mêlée | Loïc | 02/03/17 | 5j |
| Attaque à distance des ennemis | Loïc | 07/03/17 | 4j |
| Gestion des dégâts et des périodes d'invincibilités | Loïc | 12/03/17 | 3j |
| Test du jeu et correction des bugs | Loïc | 15/03/17 | 3j |
| Tutoriels JavaFx | Elsa | 04/01/17 | 15j |
| Création user et adaptation tuto | Elsa | 01/02/17 | 15j |
| UserOverViewController. | Elsa | 01/02/17 | 15j |
| User | Elsa | 01/02/17 | 15j |
| GenerateLaby | Elsa | 10/03/17 | 2j |
| AbstractFloor | Elsa | 10/03/17 | 2j |
| Correction de bugs | Elsa | 10/03/17 | 2j |
| Mise en commun du code : | Elsa | 01/02/17 | 15j |
| GameLayoutController | Elsa | 01/02/17 | 15j |
| FloorFromFile | Elsa | 01/02/17 | 15j |
| FloorView | Elsa | 01/02/17 | 15j |
| Tests et debug | Elsa | 01/02/17 | 46j |
| MoteurPhysique | Louis-Maxime | 28/01/17 | 7j |
| FloorView | Louis-Maxime | 01/02/17 | 2j |
| Création des interfaces | Louis-Maxime | 06/02/17 | 1j |
| FloorFromFile | Louis-Maxime | 06/02/17 | 1j |
| World | Louis-Maxime | 06/02/17 | 2j |
| Création de Abstractitems et Classe dérivées | Louis-Maxime | 06/02/17 | 2j |
| Animation | Louis-Maxime | 09/02/17 | 1j |
| Monnayeur | Louis-Maxime | 21/02/17 | 1j |
| Life | Louis-Maxime | 22/02/17 | 1j |

## 5.2 Figure 2 - Diagramme UML