Rapport du projet de programmtion 2

Jules Kozolinsky, Gabriel Lebouder, Julien Rixte

1er mars 2016

1 Introduction

Dans ce projet, nous avons du coder un jeu de type tower defense. Nous avons donc choisi de mettre en scène la protection d'une vallée contre des vagues de monstres ennemis. Pour cela, le joueur pourra construire des tours au dessus de la vallée afin de s'assurer que les monstres ne la traversent pas.

Nous avons subdivisé notre code en différents packages traitant chacun d'un aspect du problème. Nous expliquerons dans un premier temps le fonctionnement du jeu. Puis, nous présenterons les interactions entre nos packages. Enfin, nous verrons plus en détail l'organisation de chaque package.

2 Fonctionnement du jeu

Le jeu se déroule en différents niveaux. À chaque niveau, le joueur se voit attribuer un montant d'or et un nombre de vies. L'or permet de créer des tours. Le nombre de vies diminue lorsque des monstres parviennent à passer les défenses érigées par le joueur. Lorsque le joueur n'a plus de vie, il a perdu$_{thegame}$.

Chaque niveau consiste en une série de vagues de monstres précédées chacune d'une phase de positionnement défensif. Lors de la phase de positionnement, le joueur peut placer des tours de défense dans la limite de l'or disponible. Ces tours peuvent être de plusieurs type et certaines, bien que plus puissantes, peuvent coûter plus cher que d'autres. Une fois le positionnement effectué, le joueur peut lancer la vague de monstres en cliquant sur le bouton play.

Les monstres vont alors essayer de passer outre les défenses du joueur. Chaque monstre tué rapporte un montant d'or proportionnel à la difficulté à tuer celui-ci. Pendant cette phase de jeu, il est impossible d'ajouter les tours. C'est pourquoi nous avons ajouté un bouton d'avance rapide permettant d'accélérer le jeu.

À la fin du jeu, il suffit de cliquer sur l'image indiquant la défaite ou la victoire du joueur pour recommencer une partie.

3 Interactions entre nos packages

Nous avons choisi de diviser le code en trois parties principales : l'interface graphique (**gui**), le control du jeu (**game**) et la carte (**map**). L'idée derrière cette organisation était de cloisonner le plus possible différentes parties du codes afin qu'elles puissent fonctionner indépendemment.

Ainsi, afin que l'interface graphique puisse être changée aisément, aucun élément ayant trait à **gui** n'apparaît dans les autres parties du code. Pour cela, gui ne fait que lire les informations de **map** pour afficher les éléments de la carte et demande à **game** de s'actualiser lorsque cela est nécessaire.

TODO Un paragraphe sur les interactions entre **map** et **game**

TODO à compléter Une quatrième partie, **entities**, qui correspond à la configuration des tours et des monstres, dispose de classes permettant de créer facilement de nouveaux types de tours ou de monstres.

4 Les différents packages

4.1 Entities

Cette partie contient les définition des objets qui vont évoluer sur la map.
Ils sont subdivisés en classes comme montré ci-haut.
Les classes apparaissant plus haut sont toutes abstraites, on trouve en feuilles des arbres des classes instanciables

(Tower1, Tower2, Monster5, Tower1Type, Monster6Type...).
les classes XXX1Type sont des objects

4.2 Game

4.2.1 Round

4.2.2 Level

4.3 Parser

4.4 Map

bonjour messieurs jules et alix ça va bien ? ?
tututu

4.5 Gui

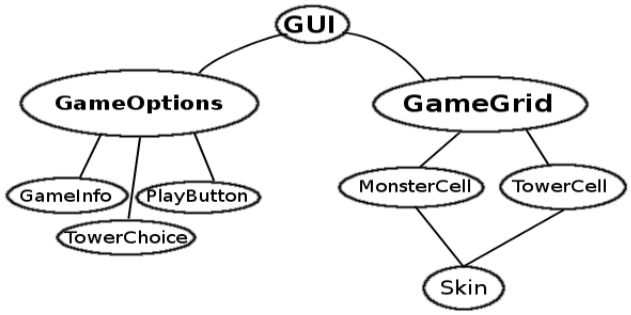

FIGURE 1 – Organisation de GUI

Le package **gui** contient l'interface graphique du jeu. Celle-ci a été écrite avec swing. L'objet principal est *MainFrameGUI*. Il contient notamment la fonction actualize qui permet d'actualiser complètement **gui**. L'interface est alors séparée en deux parties : *GameOptions* qui contient les options du jeu, situées en haut et *GameGrid* qui contient la grille du jeu. *GameOptions* est principalement constitué de boutons et d'informations sur le jeu. *GameGrid* est le coeur de l'interface. C'est une grille qui peut contenir des *MonsterCell* ou des *TowerCell*. La fonction actualize de *GameGrid* est la principale fonction d'actualisation de l'interface : c'est elle qui va lire la carte afin de placer les monstres et les tours à leur place.

La gestion des images a posé de nombreux problèmes : en effet, nous n'avons pas trouvé d'autre moyen que de recharger le fichier image pour changer ses dimensions. Nous avons donc essayé de minimiser le nombre de fois où l'on change de taille d'image. De cette idée est née la classe *Skin* qui permet un partage d'image entre les différentes entités de la grille et une évaluation paresseuse des différentes échelles dont on a besoin pour l'image. Cette classe permet donc de mettre autant de monstres qu'on le souhaite sur une même case sans pour autant ralentir de manière significative.

5 Ce qu'on aimerait faire