

Rapport du projet de programmation 2

Jules Kozolinsky, Gabriel Lebouder, Julien Rixte

18 mai 2016

1 Introduction

Dans ce projet, nous avons codé 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

Notre jeu inclue la possibilité de comprendre plusieurs level, ordonnés par difficulté (simple, medium, hard, NP-hard, EXP-hard, etc), à choisir à l'ouverture du jeu. Les level diffèrent par les rounds qu'ils comprennent et par les conditions initiales (argent et vie). Une fois un level choisi, la partie commence :

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*.

Le jeu se déroule en différents rounds. Chaque round 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 contrôle 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 indépendamment.

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.

Le package game interroge le package map. La carte se charge du déplacement physique des monstres et de l'emplacement des tours. Ainsi ce sera le package **map** qui déterminera s'il est possible de placer une tour à une position donnée, et mais aussi de la position où le monstre doit se déplacer. Le **map** dispose en effet d'une carte des monstres et des tours, alors que **game** dispose uniquement d'un *Set* de monstres.

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. C'est dans cette partie que les caractéristiques des différents objets (monstres et tours) sont définis, ainsi que les types correspondants. Le package entities définit les fonctions apply des objets qui appellent le package map pour trouver les monstres dans le cas des tours, les cases de progression dans le cas des monstres.

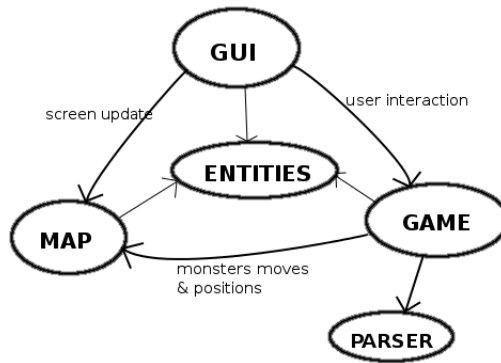


FIGURE 1 – Interactions entre les packages

4 Les différents packages

4.1 Entities

Cette partie contient les définitions 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 les classes instanciables (Tower1, Tower2, Monster5, Tower1Type, Monster6Type...).

Les classes instanciables se finissant en Type sont des objets.

Les tours ont pour caractéristiques : wait-since, pos, tower-type, frequency, priority, range, price, power, condition.

Les monstres ont pour caractéristiques : wait-since, pos, init-pos, monster-type, slowness, gold, life, orientation-h, orientation-v, speed-pix-h, speed-pix-v, pix-pos, ainsi qu'une méthode receive-dammages.

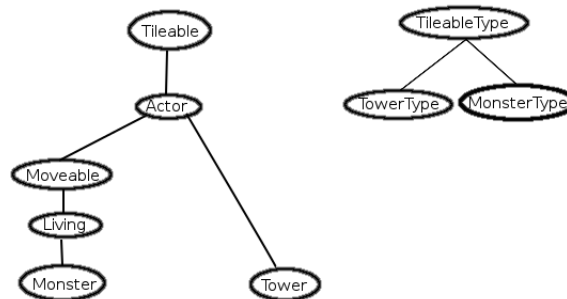


FIGURE 2 – Les différents niveaux d'entités

4.2 Game

C'est dans ce package qu'est géré le déroulement interne du jeu.

4.2.1 Round

Le package round gère les actions faites à chaque tick du timer par les tours et les monstres, il dispose pour cela d'un set de monstres et des vagues susceptibles d'arriver sur la map. C'est lui qui appelle les apply des tours et monstres. C'est donc ce package qui s'occupe de faire tirer les tours, de faire avancer les monstres et de les retirer de la map et du set quand ceux-ci meurent.

Le set nous a été nécessaire car les monstres peuvent avancer sur un tick, et nous ne pouvions donc pas faire un parcours des positions sur la map des monstres.

4.2.2 Level

Le package **level** est celui qui est appelé par **gui**, c'est lui qui relance **round** en permanence et qui gère les cas d'arrêt de jeu. C'est également **level** qui fournit à **round** la listes des vagues de monstres qui correspondent, ceci en demandant au **parser** de le faire.

4.3 Map

Le package **map** représente les emplacements physiques des entités du jeu. En réalité, il existe trois cartes différentes : *ground* représentant les obstacles fixes du jeu (par exemple des roches, des arbres ou encore des rivières), *towers* représentant la position des différentes tours, et *monsters* représentant la position des monstres pendant un round. Ces trois cartes étant *privées*, les autres packages **game** et **gui** appellent des fonctions particulières du package, par exemple s'il est possible de placer une tour, ou pour avoir accès aux monstres situés à une position donnée, ou également pour connaître la prochaine position d'un monstre.

En effet c'est le package **map** qui se charge de calculer le déplacement des monstres. Pour cela il utilise l'algorithme de Dijkstra. Les monstres sont susceptibles d'arriver depuis n'importe quelle case à gauche pour arriver jusqu'à n'importe quelle case de droite. Nous calculons alors tous les chemins possibles des cases de gauche vers celles de droite, et lorsqu'il existe plus d'un chemin de plus courte distance, les monstres choisissent aléatoirement parmi ces chemins les plus courts. Lorsqu'on place une tour, on vérifie ainsi bien qu'il existe au moins un chemin de la gauche vers la droite. Lorsqu'un monstre (le *bélier*) détruit une tour, on recalcule les itinéraires les plus courts pour les autres monstres présents sur la carte.

Pour le calcul de l'algorithme de Dijkstra, nous avons utilisé un module déjà existant composé d'une part d'un package **graph** et d'autre part d'un package **dijkstra**. Le premier définit la notion de graphe, de sommets et d'arêtes tandis que le deuxième calcule l'algorithme sur un graphe donné. Ainsi, dans le package **map** la fonction *compute_path* crée le graphe des positions à partir de la matrice *towers* et applique la fonction de calcul de l'algorithme du package **dijkstra**.

4.4 Gui

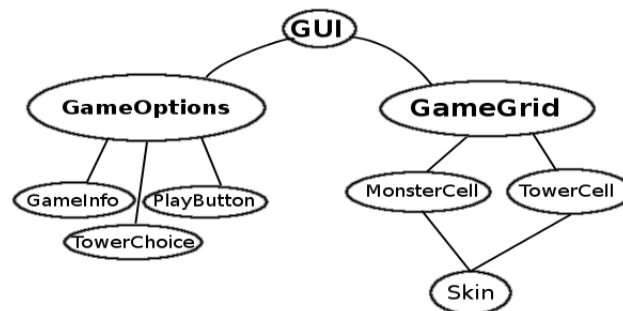


FIGURE 3 – Organisation de GUI

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. Passage au continu :

5 Ce qu'on aurait aimé faire si on avait eu davantage de temps (tri par ordre de difficulté croissante)

- Des obstacles, posables pendant un round, coûtant très chers, qui ne tirent pas et ne rapportent rien quand ils sont détruits, mais qui peuvent faire changer la trajectoire des monstres.
- On peut améliorer les tours, on peut aussi les détruire et on récupère un peu d'or.
- La puissance des tours peut décroître avec la distance.
- Un parser aléatoire fonctionnel.