U BomberMan

Thomas Morin et Enzo Sekkai

January 2021

Université de Bordeaux - *Licence Informatique - Programmation Orientée objet*

Difficultés

La plus importante difficulté que nous avons rencontré était de notifier correctement le gameEngine que des nouveaux éléments devaient être rajoutés (exemple bombes, explosions, ...). Pour ceci, nous avons opté pour un mécanisme de drapeau implémenté dans le *Game*, avec trois éléments. Le premier est la fonction *hasElementsLevelChange*, qui permet de dire au gameEngine si le monde a de nouveaux éléments depuis le dernier update ou non (hors changement de niveau). Le deuxième élément est la fonction *getNewDecors*, qui renvoie les decors théoriquement non placés du niveau actuel par le biais d'une Map. Enfin, le dernier élément est la fonction *elementsLevelChanged* qui permet de réinitialiser cette Map.

Élements à implémentations libres

Dans cette partie, nous parlerons des manières que nous avons eu de gérer certains éléments du jeu dont l'implémentation était laissée à libre interprétation.

Le premier élément laissant à confusion était la gestion des monstres n'étant pas dans le même niveau que le player. La question était devait-ils continuer à marcher ou non ? Nous avons répondu non à cette question, dans la mesure où certains monstres pour se déplacer, demandait la position du joueur. Dans le cas où le joueur n'était pas dans le niveau, cela pouvait passablement poser des problèmes. Nous avons donc choisi de ne pas faire bouger ces monstres.

Un autre élement était la possibilité pour les monstres de se marcher sur eux (ie si il pouvait y avoir plusieurs monstres sur la même case. Nous y avons répondu en disant que oui. Une modification allant dans le sens contraire serait par contre rapide, et consisterait en modifier la méthode de *game*, *positionAllowed-ToMonsters*.

Le dernier que nous pouvonss citer est la gestion des explosions sur les clés : le problème rencontré était que les clés bloquait les explosions des monstres quand ceux-ci étaient dessus. Nous avons donc implémenté un mécanisme différent : les clés n'explosent pas mais le monstre peut exploser quand il est sur la clé. Cela a pour effet d'activer le mode explosion du monstre, détaillé plus bas.

Nouveaux éléments

Dans cette section, nous allons parler des différents éléments que nous avons implémenté en plus des consignes afin de créer un jeu plus abouti.

Tout d'abord, nous avons implémenté un mécanisme de mouvement intelligents des monstres, illustrés par l'interface Automovable, dans le package *fr.ubx.poo.model.go.character.automovablepolicies*. Cette interface fournit des méthodes statiques pour générer aléatoirement la vitesse du monstre en fonction du niveau et la politique de déplacement utilisée. Afin d'implémenter une nouvelle politique de déplacement, il suffit de surcharger la méthode *sortDirections* afin d'ordonner les directions en fonction des préférences, et de créer un nouveau type d'enum dans l'enum *AutomovableType* que l'on renvoie en surchargeant la méthode *getType*. Ainsi, implémenter une nouvelle politique de déplacement est très simple. L'utilité de la méthode *getType* et du type *automovableType* est utilisé par la vue afin de savoir quel monstre afficher : en effet, nous avons créé de nouvelles images de monstres (image basique coloriée) et les Sprites utilisent ces nouvelles images. Créer une nouvelle politique ne demande néanmoins pas de créer de nouvelles images : c'est alors l'image par défaut qui est renvoyée. Nous avons en outre implémenté 5 différentes politiques, une aléatoire (*RandomPolicy*), une dont le but est de viser le joueur (*OnPlayerPolicy*), et une dont le but est de viser la position derrière le joueur afin de le piéger en sandwich (*BehindPlayerPolicy*), grâce à d'autres monstres implémentant une autre politique, visant la position devant le joueur (*InFrontOfPlayerPolicy*). Enfin, la dernière politique est semi-randomisé, et s'appelle *MCPolicy*. Son objectif est de déterminer le meilleur mouvement à faire en simulant des mouvement aléatoires du monstre et du joueur.

Un autre mécanisme intéressant est le mode explosion du monstre : Quand celui-ci explose, l'explosion est toxique, et pendant une seconde, si le joueur va dessus alors il est dommagé. Ce poison est illustré par un nuage de la couleur du monstre.

Nous avons aussi implémenté trois nouveaux bonus.

Les deux premiers sont un peu spéciaux, et leur utilisation dépend de la valeur d'une Constante, stockée dans *Constants*. En effet, nous n'avions pas la possibilité de mettre des inputs différents, mais la possibilité de les rajouter augmente grandemenent la jouabilité. Nous avons donc implémenté un mécanisme proche des directives de préprocesseurs : si la constante *INPUTMODIFIED* est à vrai, alors nous avons utilisé des inputs différents, sinon nous avons gardé les inputs classiques (entendre, la barre espace). Nous détaillerons donc dans un premier temps les deux premiers bonus implémentés puis nous détaillerons la manière de les utiliser dans les deux cas.

Le premier est un bonus mine (représenté par l'objet *Landmine*). C'est une bombe qui n'explose pas en fonction du temps, mais quand un personnage (monstre ou joueur) marche dessus ou quand un explosif explose à côté. Afin de poser cette mine, il faut récupérer un bonus landmine (représenté par une mine de couleur rouge), ce qui incrémente le compteur de landmines dans la StatusBar. Quand ce compteur est supérieur à 0, cela donne le droit au joueur de poser une mine. La mine sera alors positionnée devant le joueur, à condition que la position ne comporte pas de decor, de bonus ou de boites. Si la mine ne peut être positionnée, la mine est gardé pour le prochain appui légal. Si elle peut être posée, alors le nombre de mines disponibles est décrémenté et l'explosion d'une mine ne donne pas droit de nouveau à la mine.

Le second bonus est un bonus épouvantail (représenté par l'objet *Scarecrow*). C'est un objet qui va attirer les monstres dont la politique de déplacement est liée à la position du joueur (utilisation de la méthode *getPlayerPosition* de l'objet *Game*). C'est un objet qui peut exploser si une mine ou une bombe est posée à sa proximité. Afin de poser un épouvantail, il faut récupérer un bonus scarecrow (représenté par une image du joueur petite et rouge), ce qui donne le droit au joueur de poser un épouvantail. Si le joueur demande à le poser, l'épouvantail sera alors positionné à la position du joueur, à condition qu'il n'y ait qu'un seul épouvantail dans le niveau (plusieurs épouvantails rendrait compliqué la gestion de la position à laquelle attirer les monstres). Si l'épouvantail n'est pas positionnable, il est gardé jusqu'à ce que cela soit possible. Il faut aussi savoir que l'on ne peut stocker qu'un seul épouvantail, c'est à dire que récupérer plusieurs bonus scarecrow en ne positionnant qu'un seul épouvantail ne permettra tout de même que d'en positionner un.

Quand il est autorisé de changer les inputs, pour poser ces deux objets, on utilise des touches différentes : la mine utilise la touche SHIFT, et l'épouvantail utilise la touche CAPS (VERR. MAJ) ou la touche S (pour des problèmes de compatibilité).

Quand il n'est pas autorisé de changer les inputs, alors la gestion est un peu plus compliquée : tout se fait sur la barre espace de al manière suivante : si il y a un épouvantail de plaçable, alors on le place. Sinon on essaie de placer autre chose. Si il n'y a pas d'épouvantail de plaçable (ie si pas d'épouvantail stocké ou un épouvantail déjà posé dans le monde), alors si il y a des mines, on essaie de les placer. Sinon, on essaie de placer une bombe. Si on ne réussit PAS à placer une mine, contrairement à l'épouvantail, on ne peut PAS poser de bombes. Ainsi, pour poser une bombe, il ne faut pas avoir de mines.

Le troisième bonus implementé est un bonus infection (representé par l'object *Infected*). Cette idée vient de la majeure partie des jeux types bomberman qui implémentent un tel bonus. Ce bonus, quand il est récupéré, peut avoir des comportements différents. Ces comportements sont appelés quand le joueur effectue un mouvement. Parmi ces comportements, il y a par exemple d'effectuer des mouvements aléatoires ou encore de poser des bombes à chaque mouvement. La durée d'une infection est de trois secondes, et quand le joueur est infecté, sa teinte de couleur est changée (il devient plus rouge). L'implémentation d'une nouvelle infection est très simple et se fait juste grâce à la surcharge de la méthode *makeAction* de l'infection.

En outre, nous avons implementé un mécanisme de meilleur score, affiché à la fin. Ce système utilise un fichier local afin d'enregistrer les meilleurs scores, et donc, pour un bon fonctionnement de celui-ci, il faut, après un premier build, lancer l'application en ajoutant à gradle l'option *-x processResources* afin qu'il ne recharge pas toutes les ressources disque. Il faut noter que le score n'est enregistré que dans le cas où le joueur est gagnant. Plus bas est détaillé le fonctionnement pour que le nom du joueur soit pris en compte.

Enfin, nous avons implémenté une fonctionnalité de génération aléatoire de niveaux dans l'objet *WorldBuilder*. L'utilisation de cette fonctionnalité est décrite dans la partie suivante Nous pouvons faire quelques remarques sur celle-ci néanmoins : elle fournit un puissant moyen de jouer des parties variées, mais elle possède quelques inconvénients : par exemple, rien assure qu'il existe un chemin permettant d'aller de la position du joueur à la porte suivante ou à la princesse.

Fonctionnement du projet

Dans cette partie, nous allons décrire certains éléments de fonctionnement de notre projet.

Tout d'abord, le lancement peut se faire de deux manières. Un lancement sans argument donne droit aux niveaux déjà enregistré en local, sans aucune personnalisation. Un lancement avec arguments (pour rappel, le passage d'arguments sous gradle se fait avec *./gradlew run --args="arg1 arg2 ..."*) permet plusieurs choses. Si l'on veut faire une partie aléatoire, il faut donner l'argument **type=random**. Pour donner son nom (pour l'affichage du score par exemple), cela se fait grace à l'argument **name=yourName**. Enfin, dans le cas d'une partie aléatoire, si l'on veut préciser le nombre de niveaux, il faut passer l'argument **nbLevels=NombreDeNiveaux**. Ainsi, si Michel veut jouer 4 levels aléatoires, il le fera de la forme *--args="type=random nbLevels=4 name=Michel"*. Il faut aussi préciser que aucun argument n'est obligatoire : si pas de type précisé, ou un type non égal à random, un jeu sera chargé depuis le disque. Si pas de nombre de parties précisé dans le cas d'un jeu aléatoire, 5 niveaux seront lancés (en précision, l'argument nbLevels n'a aucune influence dans le cas d'un fichier lancé depuis le disque). Enfin, si pas de nom donné, le nom par défaut sera Moi.