Bomberman

Groupe 118:
Laritza Cabrera Alba 68171400
Magali Legast 78531500

Mai 2019

Contents

1 Introduction

La tâche pour ce projet est d'implémenter une version dans le langage Oz du jeu "Bomberman". Pour y arriver on utilise une interface graphique donnée et une structure particulière permettant d'utiliser des ports. Un des objectifs est aussi de développer de chouettes extensions. Nous avons développé des controleurs de jeu "turn by turn" et "simultaneous" qui fonctionnent avec nos joueurs basiques et "smart". Nous avons également créé un système de "map" aléatoire pour mieux tester notre programme et un player plus avancé qui n'est pas abouti.

2 Choix d'implémentation

Pour ce jeu on a deux versions : "Turn By Turn" et "Simultaneous". Pour y arriver nous avons réalisé une structure proche de Model View Controler (MVC) où on sépare la prise de décision, l'exécution des actions et l'affichage dans le GUI. Nous avons aussi créé un module nommé GameControler, à l'intérieur de celui-ci, on a imaginé une structure appelée GameState où on a fait une sorte de constructeur. Pour s'initialiser cette environnement, on utilise l'information fournie dans le fichier "Input". Dans ce GameState on garde séparément tout l'information liée au jeu, par exemple, la liste des joueurs, les variables comme Point Box et Point Bonus qui indiquent la valeur du Bonus Point. Si on voulait changer le système de cotation, ce serait très facile à faire (possible petite extension), aussi les positions de boxes point/bonus, on le mémorise dans deux listes respectivement, avec cette structure on évite de parcourir inutilement toute la Map. On dispose aussi de deux listes qui mémorisent les Players avec un état on ou off, respectivement, pour ne pas devoir vérifier à chaque tour si le Player est alive. On a d'autres chouettes information stocke qui nous permettent de maintenir une vision globale de cet GameState. Pour pouvoir modifier ce GameState, on a créé une série de fonctions et procédures qui permettent de faire un update facilement, on a créé aussi un port qui permet de partager un même GameState avec différents Players, pour la partie simultané par exemple. On peut la voir comme une librairie qui peuvent être utilisé par le Main et les Bombers. Pour pouvoir arriver à faire cette structure MVC on crée dans la GameControler une fonction DoAction qui demande au Bomber l'action qu'il veut faire mais le bomber n'update pas son information tant au'il n'a pas reçu un message info avec cette action de lui-même, avec ce système-là on peut séparer l'action de l'exécution, donc la fonction *DoAction* génère une séquence de message qui ne s'exécutent pas jusqu'à ce qu' on appelle une fonction qui analyse ces messages et les exécute, ensuite on fait un broadcast et après on envoie au GUI l'info pour montrer l'info. On a choisi de faire la mise à jour de l'état interne du Bomber au moment où il reçoit les messages qui ont été broadcastés à tous les Players car on devait rester consistant dans la structure MVC et aussi utiliser le design qui a été donnée dans le base du projet.

2.1 TurnByTurn Controler

Dans le Turn By Turn on crée dans la Main une fonction *TurnByTurn* qui fait une initialisation du jeu, crée les players et on crée un environnement (notre GameState) un fois créé on commence à tourner en boucle avec la fun *LoopTurnByTurnGameStatePlayersList* on défini un tourné avec un parcours de la Liste de Players, et quand la tournée est fini on recommence mais avec un Liste actualisé de players qui ne sont pas dans un etat off. Dans ce boucle On fait trois actions fondamentaux, Check if alive, Update le GameState(vérifier l'état de bombes, s'il faut les exploser ou pas et check les damages qui a occasionné la bombe) un fois Updatede on Fait un Action Avec ce Gamestate Updated et ici comme explique précédemment on utilise la fun DoAction que sépare le action de l'exécution, donc on fait l'appel à cette fonction on génère un série de messages que après on les exécute et cette exécution update La Gamestate ensuite on le broadcast et on le montre et on continue a boucler. Jusqu'à que les conditions de GameOver sont atteintes.

2.2 Simultaneous Controler

Dans la partie simulateous, on fait la même intilisation que pour la partie TurnByTurn, mais ici on crée un port pour ce GameStatete afin que tous les players et l'update de Bombes puissent accéder au même State du Game. Ensuite on crée un thread pour update les Bombes et vérifier les dommages et un autre pour chaque Player. Dans le thread où on update les bombes on vérifie

dans ce Game State générale en commençant par parcourir une liste de Bombes et vérifiér si le time (un time random entre le TimingBombMin et TimingBombMax) qui a été donnée pour exploser la bombe est déjà atteint. Si la première bombe de la liste n'a pas explosée on fait rien, et on continue en boucle jusqu'à atteidre l'expiration du timer. On vérifie toujours les conditions de Game Over si c'est Game Over on arrête de vérifiér. Comme il dispose aussi d'un thread pour chaque player et tous ont accès au même GameState et qu'on a aussi une structure des actions séparés de l'execution on a fait deux blocages qui nous permettent d'être plus efficient et en pas devoir bloquer l'update de bombes inutilements. Ces deux zones critiques sont les suivantes.

1. On entre dans la première zone quand on arrive a avoir l'accès au GameState entre les players. Si on a l'accès on évite que les autres players fassent un update du GameState, mais les bombes elle peuvent continuer a update. Ensuite, on fait un copie de State et un DoAction qui nous retourne une liste d'actions mais comme on a rien modifié dans le State partagé, tout va bien, quand on a cette liste et aussi que le time random entre le ThinkMin et le ThinkMax a été atteint on demande l'accès à Update Bomb pour modifier le Gamestate Général on entre dans la deuxième zone critique.

2. Dans cette deuxième zonne ni les autres players ni l'update de bombes peuvent modifier le GameState, donc on est dans une zone critique où on peut tout faire sans problèmes. Comme on a obtenu la liste d'actions d'une copie du State qui n'a pas été actulisé par les Bombs on check alors deux conditions : si ce n'est pas GameOver et que je ne suis pas mort (état off) alors j'exécute ma liste d'actions qui vont Update le GameState, ensuite broadcast et après send to GUI, si c'est GameOver ou que je suis mort (état off) je fais rien et on sort en débloquant les threads pour que les autres puissent continuer.

L'objective de ce double blocage est de ne pas permetre aux autre players de modifier le GameState mais premetre que le bombes explosent dans cet periode de temps et potencialement tuer un player qui est en train de prendre un decision et que n'a l'a pas encore executé. Pour pouvoir arriver a faire ça on a pris le decision de update le player au momment qu'il recçoit les info de broacast et pas au moment ou je lui dit de faire une action, mais ça peut etre un problemme d'interoperabilité avec les autres groupes qui ont fait une implemenation de leur players avec un update dans le moment où je l'envoie a faire une action.

3 Players

3.1 BasicPlayer

Notre joueur basique a un comportement aléatoire qui lui permet de réagir en toute situation. Dans 90% des cas, il essaye d'effectuer un mouvement pour se déplacer. Si la première direction qui est choisie de manière aléatoire n'est pas possible, c'est à dire que la case en question est un mur où porte une boîte, une autre direction est choisie. Si aucune direction n'est possible, par exemple si la position de spawn du joueur est entouré de murs et de boites, le joueur dépose une bombe pour pouvoir sortir par la suite. Dans les 10% de cas restant, le joueur essaye de poser une bombe. S'il ne peut pas le faire (pas de bombe en réserve où présence d'une de ses propres bombes sur la case), il se déplace.

Pour permettre ce fonctionnement, notre joueur basique enregistre toutes les informations du jeu, par rapport à lui-même et à l'état du jeu, dans une structure qui est passée en argument à chaque appel de la fonction TreatStream. Il s'agit du tupple Info ci-dessous:
Info = infos(id:ID lives:NbLives bombs:NbBombs score:Score state:State currentPos:CurrentPos initPos:InitPos map:Map rivals:Rivals)
où ID est l'id de type ¡bomber¿ du joueur, NbLives et NbBombs le nombre de vies et de bombes qui restent au joueur, State son état ('on' ou 'off'), CurrentPos sa position actuelle, InitPos sa position de spawn. Map est une représentation actualisée du plateau de jeu qui permet au joueur de savoir à tout moment le contenu d'une case, hormis la présence de bonus dont notre player n'utilise pas la localisation. Les différents éléments qui peuvent se retrouver sur une case sont représentés par des nombres de grandeurs différentes qui s'additionnent. L'ordre des milliers représente le nombre de bombes adverse, l'ordre de la centaine la présence d'une bombe du joueur en question, l'ordre de la dizaine le nombre de joueurs et l'unité la présence ou non d'un mur ou d'une boite. Cette

manière d'enregistrer l'information permet de vérifier facilement la présence d'un élément déterminé à l'aide des opérations 'div' et 'mod' tout utilisant la structure map de départ et en limitant la place que cette carte prend en mémoire. Enfin, le champ rivals contient une liste des états de tous les players adverses enregistrés dans un tupple Rival = rival(id:ID state:State pos:Pos) où ID est l'id de type ¡bomber¿ du joueur rival en question, State son état 'on' ou 'off' et Pos sa position actuelle. A chaque changement de l'état ou de la position d'un joueur adverse, notre player ajoute cet état actualisé à la liste. Notre player peut donc toujours connaitre les informations les plus récentes d'un joueur rival elles se trouvent dans le premier élément de la liste avec l'ID recherché.

Certaines informations propres au joueur telles que sa position de spawn, son nombre de bombes et son score sont mises à jour lorsque le joueur reçoit les messages assignSpawn(Pos) et add(Type Option ?Result). Toutes les autres informations sont mises à jour lorsque le joueur reçoit le message info(Message) correspondant. Dans notre implémentation, il est nécessaire d'attendre de recevoir ces messages broadcastés à tous les joueurs pour effectuer ces mises à jour. Dans le cas contraire, si les informations sont mises à jour lorsque les messages envoyés à un unique player,

Les informations enregistrées sont très complètes et notre basique player ne les utilises pas toutes. Il s'agit d'un choix de conception motivé par le fait que ce joueur est destiné à servir de base pour d'autres joueurs plus avancés. Il nous semble donc intéressant d'avoir une base comportant déjà toutes les informations nécessaires pour pouvoir implémenter facilement différentes fonctionnalités par la suite sans devoir créer de nouvelles structures ou de nouvelles manières d'enregistrer de l'information pour chacune des fonctionnalités.

3.2 Survivor Player

Le joueur "survivor" essaye de se retirer de la trajectoire d'une bombe et choisit une des directions les plus sûres lors de tous ses déplacements. Il dépose des bombes uniquement lorsqu'il se trouve sur une case sans danger, c'est à dire hors de portée de toute bombe. Le reste de son comportement est aléatoire et, lorsqu'il se trouve sur une case sans danger, il essaye de déposer une bombe dans 10% des cas.

Il a été créé comme une amélioration du joueur basique. Il utilise les mêmes structures et globalement la même manière de fonctionner que ce dernier. La seul différence dans l'implémentation est sa manière de réagir à la réception du message doaction(?ID ?Action), ce qui engendre le comportement explicité ci-dessus. Pour éviter les cases dangereuses, le joueur évalue leur dangerosité sous forme d'une valeur qui correspond au nombre de bombes qui peuvent toucher le joueur s'il se trouve sur cette case-là. S'il y a une égalité entre plusieurs case avec le niveau le plus bas, le joueur en choisis une au hasard.

4 Interoperabilité

4.1 Interoperabilité

Sauf indication contraire dans les détails d'un test particulier, les tests d'interopérabilité ont été effectués sur des map de taille 7x13 générées aléatoirement. A l'état initial, ces maps comportaient toutes 4 zones de spawn, des cases avec des murs, des cases avec des boites et des cases vides. Aucunes des maps générées pour les tests ne présentaient de problème structurels pour le jeu, tels qu'un joueur dont la position de spawn serait entouré de murs par exemple, ou alors ces situations n'ont pas été prises en considération pour les tests. Le nombre de joueurs utilisés était de 2 et le nombre de vie et de bombes initiales respectivement 5 et 2.

4.1.1 Players basiques

Nous avons testé deux players basiques, Player000John, groupe 3 (Jérôme Van Der Elst et Nicolas van de Walle) et Random, groupe 55 (Gildas Mulders et Lylia Semerikova) Il s'agit de deux joueurs au comportement aléatoire qui se comportent tous les deux de manière attendue. Nous n'avons remarqué d'action illégale pour aucun des deux. Ces deux joueurs sont interopérables avec notre propre joueur random et le mode de jeu turn by turn.

Dans le mode simultaneous, les players John se comportent bien de manière simultanée. Le joueur random, par contre, donnait plus une impression de "pseudo turn by turn" parce que les

mouvements étaient principalement effectués à tour de rôle. Par ailleurs, nous observés également deux problèmes, identiques pour les deux players. Premièrement, il est arrivé que le jeu s'arrête alors que tous les joueurs ont encore des vies et qu'il reste des boites. Nous suspectons que ce problème vienne d'une actualisation de l'état du player et du jeu à la réception des messages doaction et autres messages envoyés à un player particulier, ce qui peut bloquer une thread. Cette théorie a été vérifiée pour le player random, mais nous n'avons pas pu l'infirmer ou la confirmer pour le player John. Deuxièmement, deux players sont toujours annoncés gagnants alors qu'un seul player a gagné. Nous avons remarqué qu'il ne s'agissait pas d'un problème d'interopérabilité et nous avons pu le comprendre et le corriger par la suite.

4.1.2 Players smart

Nous avons testés 3 players avancés. Le player Player001name du groupe 55 (Gildas Mulders et Lylia Semerikova) est essaie d'éviter les bombes et d'aller chercher les points. Le player Player100Advanced du groupe 100 (Edgar Gevorgyan et Louis Navarre) se dirige normalement vers les boîtes s'il y en a à proximité et et évite les explosions. Le player Player038Luigi du groupe 38 (François de Keersmaker et Margaux Gérad) qui est supposé éviter les bombes, placer des bombes s'il est près d'une boite et aller chercher les points.

Au cours de ces tests, un comportement anormale a été observé chez les trois players et dans les deux modes de jeux. Ils ont tendance à déposer des bombes au début du jeu, à se déplacer et ensuite à rester fixés dans un va et vient entre quelques mêmes cases. Ce comportement n'a pas été observé lors d'autres tests d'interopérabilité de ces mêmes players avec d'autres controlleurs de jeu. Nous pensons que ce problème est causé en partie par une mauvaise gestion des messages d'informations. Nous n'avons pas réussi à résoudre ce problème. Nous pensons que pour le joueur Player001name, un autre facteur influence ce mauvais comportement, à savoir que son algorithme soit prévu pour la map donnée initialement, ou une map qui lui ressemble, et qu'il est moins performants dans les maps aléatoires avec des case qui sont déjà vides à l'origine et où le joueur n'est pas toujours à proximité d'une boite ou d'un mur. En effet, avec la map donnée à l'origine, le comportement problématique se produit plus tard et moins souvent qu'avec nos map aléatoires. La situation inverse (performance meilleure dans notre map par rapport à la map initiale) est observée pour le playeur Player100Advanced.

Au niveau des comportements particuliers, le joueur Player001name évite peu les bombes qu'il pose et revient parfois vers elles alors qu'il est possible de se mettre à l'abri. Il ignore les boites et les points qui ne se situent pas sur son chemin, même si elles se trouvent sur des cases adjacentes. Le joueur Player100Advanced évite efficacement les bombes qu'il pose lorsque c'est possible. Il se dirige correctement vers les boites, mais ne va pas forcément chercher les points qu'il vient de libérer. Le player Player038Luigi évite très bien les bombes et se dirige majoritairement vers les bombes et les points.

Comme les joueurs basiques que nous avons testés, ces joueurs avancés mettent à jour leur état au moment où ils effectuent une action et non au moment où le message "info" lui correspondant arrive. Le problème de thread bloquée observé dans le mode simultané se produit aussi pour ces joueurs.

Nous avons par ailleurs également remarqué un problème important chez le joueur Player100Advanced qui faisait parfois crasher l'application en renvoyant l'erreur : Floating poit exception (core dump). Notre test a permis de découvrir cette erreur qui ne s'était pas produite avec d'autres controller. Elle était due à une division par zéro et a pu être corrigée. La nouvelle version corrigée de ce player, Player100Advanced, est également jointe à notre travail.

5 Extension

5.1 OziMax

L'extension principale n'a pas su être terminée dans les temps et n'est pas compilable, mais nous vous présentons néanmoins l'état de notre avancement de ce super smart Bomber (OziMax). Ce super Smart Bomber utilise la librairie de GameControlerBomber pour avoir son propre état interne. Nous avons créé un fonction (Thinking) qu'on appelle à chaque fois que l'on reçoit un message de DoAction, cette fonction thinking, la première chose qu'elle va faire c'est créer une copie de l'état

actuel. Son objectif est de choisir la prochain action : soit mettre une bombe soit bouger dans une des positions admissible. Pour choisir la meilleure action on utilise l'algorithme de MinMax, cet algorithme va pouvoir descendre dans un nombre max de niveaux N qu'on pourrait faire, un grand débat est le choix du N, qu'on ne va pas discuter ici. Chaque fois qu'on descend d'un niveaux on va simuler toutes les actions possibles, un peu à l'image d'un jeu d'échecs, on réfléchit un certain nombre de niveaux possibles si je fais une action x, donc ici c'est la même chose chaque fois que je descends un niveaux je vais modifier mon état comme si j'étais en train de jouer vraiment mais en commençant avec un copie de l'état actuel. Ensuite comme ma structure je peux décider de ne pas envoyer de messages de broadcast et ne pas afficher dans la Windows car tout est séparé. À chaque niveau que je descends j'ai une liste possible d'actions et je change aussi le player qui le corresponde jouer quand cette fonction arrive aux dernier niveaux N ou un ramdom time entre TimingBombMin et TimingBombMax je retourne une valeur qui sera calculée par un fonction d'évaluation. Cette fonction va donner un score pour chaque situation possible dans le State par exemple si je suis proche d'une bombe un score négatif pareillement si je suis dans une position ou il y a un point proche c'est un score positif, si j'ai déjà une certaine quantité de point c'est un score positif également. donc la fonction d'évaluation c'est d'un certain faisons la fonction qui donne la décision. À chaque étape je retourne la multiplication du score qui me donne le meilleur chemin et je multiplier par -1 pour chaque niveaux car l'autre player va aussi choisir ça meilleur possibilité donc quand je reviens j'ai choisi le meilleur score pour chaque action et c'est l'action final que je vais retourner dans le message reçu doAction (?ID ?Action).

5.2 RandomMap

On a crée une fonction de génération qui nous donne une map avec un nombre random de boxes point/bonus, floor, floor with spawn, wall (le nombre de boxes générées est plus grand que le nombre de floor with spawn). Si la condition d'extension est mise à true, on utilise ce Random Map, sinon on utilise une Map par défaut donné dans le Input File.

6 Conclusions

On peut dire que même s'il y a eu des difficultés dans la réalisation de notre projet, on est globalement satisfaits. En effet, on a fait tout ce qu'on devait faire pour arriver a avoir une bonne structure qui est flexible, consistante et qui permet de supporter des changements sans devoir tout refaire.