LINGI1131 - Rapport de projet

Louis Navarre

March 2019

1 Introduction

Ce rapport concerne le projet pour le cours LINGI1131 - Concepts de langages de programmation (*Computer languages concepts*).

Dans un premier temps, en ce qui concerne la partie obligatoire, nous avons effectué tout ce qui était demandé. Des explications sur la structure de notre implémentation de notre contrôleur et de nos joueurs concerne la première partie de notre rapport. Dans un second temps, nous avons ajouté quelques extensions à notre travail. Tout d'abord, nous avons ajouté les deux bonus supplémentaires conseillés lorsqu'un joueur en ramasse un sur le terrain de jeu: le bouclier et la vie supplémentaire. Ensuite, nous avons modifié à notre guise le `GUI` afin de le rendre plus ludique et plus approprié à un jeu lugubre et mortel tel que le *Bomberman*. **AJOUTER ICI LES AUTRES BONUS EFFECTUES!**.

2 Contrôleur `Main`

Notre contrôleur est séparé en deux parties: la gestion des évènements et la gestion des joueurs. Faire de la sorte nous permet d'éviter de dupliquer du code pour le mode de jeu tour par tour, et le mode simultanée. Ainsi, les seules différences notables entre ces deux modes de jeu se situent au niveau de la gestion des joueuers - la gestion des évènements est donc identique peu importe le mode de jeu. Toutes les fonctions/procédures référencées dans cette section sont relatives au fichier `Main.oz`.

2.1 Gestion des joueurs

2.1.1 Mode tour par tour

Le mode tour par tour est totalement géré par la procédure `TurnByTurn`. Les arguments de cette procédure sont:

- `ThePlayersPort`: la liste de joueurs à encore traiter dans l'itération.
- `TheBombs`: les bombes posées et non encore explosées.

Cette procédure "itère" (en effectuant des appels récursifs) sur la liste des ports des joueurs présents dans la partie (même les joueurs décédés). Lorsque l'itération est terminée (i.e. le reste de la liste à traiter est `nil`), la procédure vérifie l'état des bombes en attente d'explosion, et traite les bombes devant exploser à ce tour, comme expliqué dans le paragraphe correspondant ci-dessous. Ensuite, la procédure effectue un appel récursif effectue un appel récursif **C'EST PAS LE CAS, A CHANGER DANS LE CODE** si la fin de partie n'est pas détectée; et recommençant l'itération depuis le début de la liste de ports de joueurs. Les deux paragraphes ci-dessous détaillent le fonctionnement de la procédure en fonction de l'état de l'itération.

Pour un joueur. Si le joueur est toujours en vie (2.2.2), alors nous demandons au joueur d'effectuer sa prochaine action. S'il s'agit d'un déplacement (`move`), alors nous vérifions regardons si le joueur vient de récupérer un point/bonus. Nous traitons le point/bonus, et mettons à jour la carte (*map*) (2.2.1). S'il s'agit d'une bombe, alors nous ajoutons la position de la bombe plantée, l'identifiant (`ID`) du joueur l'ayant posée, ainsi que le nombre de tours avant explosions (`Input.timingBomb`).

En fin de tour. Lorsque le tour est fini (i.e. tous les joueurs ont effectué leur action), la procédure vérifie l'état des bombes: nous décrémentons les *timers* de chacune des bombes posées, et nous procédons à l'explosion des bombes où ce *timer* est nul (2.2.3).

2.1.2 Mode simultanée

Le mode simultanée est initialisé par la procédure `SimultaneousInitLoop`. Le seul argument de cette procédure est la liste des ports des différents joueurs. Pour chacun de ceux-ci, la fonction crée un *thread* qui exécute la procédure `APlayer`, prenant comme seul argument le port du joueur qu'il contrôle. Une fois cette procédure lancée, le principe est presque identique que pour l'action d'un joueur dans le mode tour par tour. La première différence est que, comme le mode de jeu l'indique, un joueur est indépendant des autres; ainsi, la procédure se rappelle elle-même pour demander au joueur d'effectuer sa prochaine action, etc...

La différence majeure se situe au niveau la gestion de l'amorçage de bombes. Ici, lorsque la procédure reçoit comme action du joueur de poser une bombe, un *thread* indépendant est créé. Le but de ce *thread* est simple: afficher la bombe à l'écran, attendre un délai fixé par les propriétés de la partie (`Input.oz`), enlever la bombe de l'écran, et traiter l'explosion de la bombe (2.2.3).

2.2 Gestion des évènements

Mot au lecteur. Les justifications qui suivent sont justifiées pour le mode simultanée, pas pour le mode tour par tour. Il était en effet possible de ne pas utiliser des agents pour le mode tour par tour, contrairement à ce qui est fait ici. Mais, dans un souci de clarté et afin de pouvoir réutiliser du code déjà existant (sans pour autant impacter les performances de notre programme, du moins estimons-nous), nous avons utilisé le même fonctionnement pour le premier mode de jeu.

Plusieurs évènements peuvent arriver pendant la partie: une boite peut exploser dû à une bombe qui a elle-même explosé, faisant apparaitre un bonus ou un point; un joueur peut se déplacer, ou encore mourir, ce qui peut potentiellement indiquer la fin de la partie. Dans ce cas, il faut savoir identifier le vainqueur, etc... Il faut donc un mécanisme qui permet aux différents agents de traitement de joueurs (2.1.2) d'être tenu au courant des dernières modifications.

2.2.1 Gestion de la carte

Cet agent est responsable de la carte de jeu. Dès qu'une modification doit être faite, un message est envoyé et traité par l'agent. Celui-ci tient donc comme argument, en plus du *Stream* à traiter, la version la plus récente de la carte. Il est également possible (et nécessaire) de demander à cet agent la dernière version de la carte de jeu, par exemple pour savoir si le joueur est arrivé sur une case où un bonus est présent, ou bien si l'explosion d'une bombe brise une boîte à bonus/point.

2.2.2 Gestion des morts

Cet agent tient une liste de tous les joueurs encore en vie. Cela permet entre-autres de déterminer si la partie doit se finir puisqu'il n'y a plus qu'un/aucun joueur présent sur la carte. Cet agent peut aussi donner la liste des joueurs encore en vie; c'est ce qui est utilisé par les agents de joueurs pour déterminer si le joueur est encore en vie ou non dans la partie.

Remarque. Utiliser l'état du joueur (*State*) pour déterminer si celui-ci était encore dans la partie n'était pas suffisant, puisqu'au cas où celui-ci venait de mourir mais n'était pas éliminé du jeu (i.e. avait encore des vies), cela provoquait une mauvaise interprétation et pouvait amener à l'élimination du joueur alors que ce n'est pas le cas.

Une valeur booléenne est "envoyée" (par liage de l'identificateur envoyé dans le message) à l'émetteur du message, pour indiquer s'il s'agit de la fin de la partie ou non. En cas de fin de partie, le vainqueur est déterminé par ses points (2.2.5).

2.2.3 Gestion des bombes

2.2.4 Gestion des positions

2.2.5 Gestion des points