

LINGI1131 - Rapport de projet

Edgar GEVORGYAN 2018-16-00

Louis NAVARRE 1235-16-00

Avril 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 concernent 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 *Bombberman*. Nous avons créé une petite carte en l'honneur de Oz. Enfin, nous avons aussi créé notre propre suite de tests, qui teste quelques cas triviaux et particuliers du comportement attendu des joueurs.

Remarques. Précisons dès maintenant que le *thinking delay* est inclus dans notre contrôleur principal et pas dans chacun de nos joueurs. Lorsque nous avons appris la réelle signification de cette règle, nous avons déjà effectué tous nos tests d'interopérabilité. De plus, **pour démarrer le jeu**, il faut appuyer sur le bouton *Start Game* en haut à gauche de la fenêtre de jeu.

2 Contrôleur Main

Notre contrôleur est séparé en deux parties: la gestion des événements 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 simultané. Ainsi, les seules différences notables entre ces deux modes de jeu se situent au niveau de la gestion des joueurs - la gestion des événements 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 encore à traiter dans l'itération.
- `TheBombs`: les bombes posées qui n'ont pas encore explosé.

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 si la fin de partie n'est pas détectée; et recommence 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 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 décrémentation 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 simultané

Le mode simultané 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 après un certain délai pour demander au joueur d’effectuer sa prochaine action, etc...

La différence majeure, elle, 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 fondées pour le mode simultané, 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 les deux modes de jeu.

Plusieurs évènements peuvent arriver pendant la partie: une boîte peut exploser dû à une bombe qui a elle-même explosé, faisant apparaître 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 permette aux différents agents de traitement de joueurs (2.1.2) d’être tenus au courant des dernières modifications.

Nous avons décidé de créer un port différent pour chaque type d’évènement, et une fonction différente pour traiter chacun des *streams* de ces ports, pour pouvoir gérer chaque évènement et chaque changement d’état du jeu indépendamment du reste. Nous expliquons ci-dessous les différents ports utilisés pour gérer ces changements d’état/évènements.

2.2.1 Gestion de la carte: MapHandler

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 détruit une boîte à bonus/point.

2.2.2 Gestion des morts: CheckEndGameAdvanced

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: BombHandler

Cet agent est utilisé lorsqu'une bombe doit exploser. Il propage le feu sur la carte, et vérifie s'il s'agit de la fin de la partie (c'est-à-dire que les dernières boîtes sur la carte ont été détruites, ou qu'il n'y a plus qu'un/aucun joueur sur la carte). Pour propager le feu, cet agent appelle la procédure `PropagationFireSimult` avec deux arguments majeurs:

- `BombPosition`: l'emplacement de la bombe
- `TheMap`: la version la plus récente de la carte

Cette fonction s'occupe de la propagation du feu. Elle affiche tout d'abord à l'écran du feu à l'endroit de l'explosion de la bombe, puis propage ce feu dans les quatre directions cardinales par rapport à cette position (avec la procédure `PropagationInOneDirection`). Cette procédure vérifie si le feu peut se propager (c'est-à-dire si il ne s'agit pas d'un mur et si la limite de propagation n'est pas atteinte). Si c'est le cas, alors nous regardons si la case de propagation est une case boîte. Si oui alors la boîte est détruite (2.2.6) et nous vérifions s'il s'agit de la fin de la partie. Si non, alors le feu à cette position peut potentiellement tuer un joueur (procédure `ProcessDeath`) et continue de se propager.

2.2.4 Gestion des positions: PositionsHandler

Le fonctionnement de cet agent est fort semblable à celui de la gestion de la carte: il tient une liste de la position actuelle de chacun des joueurs; et modifie celle-ci lorsqu'un joueur effectue un déplacement.

Cet agent est utilisé pour connaître la position actuelle des joueurs lorsqu'une bombe explose. Il était aussi possible de demander au joueur sa position (et stocker celle-ci comme argument dans la procédure `APlayer` (2.1.2)), mais nous avons préféré garder le même fonctionnement que pour les autres agents utilisés.

2.2.5 Gestion des points: PointHandler

Les points actuels sont stockés et modifiés via cet agent. Cela permet, en fin de partie, de récupérer le nombre de points de chaque joueurs ayant commencé la partie. Comme pour la gestion des positions (2.2.4), nous aurions pu stocker cette donnée dans la procédure du joueur (ou en argument de la procédure `TurnByTurn` en mode tour par tour). Notons que nous ne donnons qu'un seul et unique vainqueur par partie, même en cas d'égalité de points.

2.2.6 Gestion des boxes: BoxHandler

Il est important de tenir le compte du nombre de boîtes encore intactes sur la carte, puisque la partie est considérée comme terminée lorsqu'il n'y a plus aucune de celles-ci sur celle-ci. L'agent tient donc un compte du nombre de *boxes* encore sur la carte, et peut recevoir des notifications d'une explosion de boîte. Avant de décrémenter le compte total de boîtes encore présentes, l'agent vérifie si la boîte indiquée par la position donnée est encore sur la carte. Cela peut permettre d'éviter certains problèmes de désynchronisation entre deux explosions de bombes dans le mode simultané. C'est d'ailleurs cet agent qui demande à l'agent de la gestion de la carte de modifier celle-ci (2.2.1).

3 Joueurs : PlayerXXXname.oz

Dans cette partie du projet, nous avons mis en place 3 joueurs. Un joueur simple qui agit de manière assez aléatoire et deux joueurs un peu plus intelligents.

3.1 Joueur basique : Player100random.oz

Ce joueur se déplacera sur le plan de jeu de manière aléatoire et posera, si possible, une bombe avec une probabilité de 0.1.

3.2 Joueur avancé: `Player100advanced.oz`

Étant donné que le gagnant est le joueur possédant le plus de points à la fin de la partie, la stratégie de notre joueur avancé est de se concentrer sur les boîtes de points/bonus et non sur l'adversaire. Le joueur ne prête donc aucun intérêt aux joueurs adverses, mais uniquement à sa sécurité et aux points potentiels de la carte de jeu. Nous pouvons voir le comportement de notre joueur comme une suite de vérifications "d'états" pour le prochain mouvement à effectuer:

- Si le joueur est positionné sur une case adjacente à un bonus/point libéré, le joueur se déplacera vers cette case pour engranger le bonus/point.
- Si le joueur est positionné sur une case adjacente à une *boxe*, et qu'il a encore des bombes en stock, alors celui-ci pose une bombe.
- Si le joueur n'est pas en *safe place* (lieu sûr), alors
 - Si une case sûre est adjacente à la case du joueur, le joueur se déplace sur cette case.
 - Sinon, le joueur se déplace dans la direction opposée à la case où est située la bombe causant l'insécurité du joueur.
- Si le joueur est en lieu sûr, mais qu'il n'y a aucun(e) bonus/point/boite, alors le joueur se déplace de façon aléatoire.

3.3 Jon Snow: `Player100JonSnow`

Ce joueur a un comportement fort similaire au joueur précédent. Le seul changement majeur est que, lorsque Jon peut se déplacer sur plusieurs cases sûres, il va de préférence aller vers la case le rapprochant du/de la bonus/point/boite le/la plus proche de lui. Pour faire cela, nous effectuons un BFS où les sources sont toutes les cases sûres où Jon peut se déplacer. Ensuite, nous cherchons le/la bonus/point/boite le/la plus proche, et le joueur se déplace dans cette direction.

4 Interopérabilité

Nous avons effectué des tests d'interopérabilité avec plusieurs groupes différents. Grâce aux plateformes déployées par d'autres étudiants, nous avons pu vérifier si notre contrôleur fonctionnait avec ces autres joueurs. Pour chacun des joueurs testés qui fonctionnaient avec notre contrôleur, nous avons aussi lancé notre propre suite de tests.

- Nous n'avons eu aucun problème avec les joueurs suivants. Ils ont également passé tous nos tests: `Player021IA2` (groupe 21), `Player005Umberto` (groupe 05), `Player106noob` (groupe 106), `Player007Zorro` (groupe 07), `Player022Immortal` (groupe 022), `Player001Turing` (groupe 01), `Player001Tao` (groupe 01), `Player055Clever` (groupe 55).
- Aucun problème non plus pour les joueurs suivants, mais ceux-ci n'ont pas passé tous nos tests (en tout cas au moment où nous avons testé ces joueurs): `Player105Alice` (groupe 105), `Player007Zorro` (groupe 07), `Player010IA` (groupe 10).
- Impossible de faire fonctionner le joueur suivant (en tout cas au moment où nous avons testé ce joueur): `Player022smart` (groupe 22).

Les groupes suivants ont testé nos 2 joueurs intelligents, et nous ont donné un *feedback* positif sur ceux-ci: 01, 07, 10, 21, 22, 55, 105.

Le groupe suivant nous a indiqué que nos joueurs intelligents retournaient un *floating point exception*: 188 (Magali Legast & Laritza Cabrera Alba). Nous avons pu corriger notre erreur grâce à eux.

5 Extensions

Trois types d'extensions ont été ajoutés à notre travail:

- Bonus: Comme dit dans l'introduction, nous avons ajouté les boucliers et les vies additionnelles comme gain des boîtes "bonus" de la carte.
- GUI: nous avons modifié à notre guise le **GUI**. Chaque joueur a une petite image; des icônes pour les bombes, le feu, les bonus et points ont été ajoutées; le terrain de jeu a aussi été changé. Nous avons aussi ajouté un bouton *Start Game* qui démarre le jeu; celui-ci ne commence pas si personne n'appuie sur le bouton.
- Tests: nous avons créé une suite de tests (**Test.oz**). Cette série de tests vérifie d'abord quelques opérations triviales:
 - Faire un déplacement dans une zone autorisée
 - Recevoir des points/bombes et retourner le bon compte total
 - Mourir et réapparaître au bon endroit (spawn de départ)
 - Recevoir les vies additionnelles; recevoir un bouclier et l'utiliser lorsque le joueur a été notifié comme touché

En plus de cela, nous avons aussi testé un *border case*:

- Le joueur est déjà mort et est à nouveau notifié de sa mort: il doit *bind* les identificateurs reçus à `null`

Pour rendre son utilisation plus simple, nous avons ajouté les commandes suivantes dans le *makefile* :

- **test**: compile les fichiers nécessaires pour les tests et lance ces derniers
- **compileTest**: compile les fichiers nécessaires pour les tests
- **run_test**: lance les tests

Remarque. Pour utiliser les tests, il est demandé de mettre en commentaire les lignes 41, 42 et 44; et de décommenter les lignes 46, 47 et 49.

- Autres: Spécifions que lorsque les extensions sont activées, le délais entre deux déplacements de joueurs est diminué et est constant à une valeur de 500ms. De plus, notre joueur avancé **Player100advanced** peut à ce moment-là rester sur une case sans effectuer de déplacement. Cela permet à nos joueurs avancés d'optimiser leur stratégie de survie.

6 Les problèmes rencontrés

Nous avons rencontré deux types de problèmes majeurs lors de l'implémentation du projet. Le premier est la manière de récupérer l'état des joueurs, comme leur position ou leur nombre de points. De notre point de vue, il aurait été plus simple de pouvoir demander au joueur de donner sa position actuelle/son nombre de points, puisqu'il stocke ces informations de toute manière. Or, ici, nous devons les stocker dans le joueur et dans le contrôleur.

Le deuxième problème majeur que nous avons rencontré est la gestion de la carte. En effet, celle-ci change au cours de la partie (explosion d'une boîte, un bonus récupéré par un joueur,...) et nous avons pris du temps avant de nous rendre compte qu'il était par exemple possible d'utiliser un agent spécifique pour la carte uniquement. Nous avons encore eu quelques soucis de synchronisation entre les joueurs et le contrôleur. Par exemple, il arrivait de considérer un joueur comme mort définitivement, alors qu'il avait simplement péri à cause d'une bombe, mais avait encore des vies et n'avait pas encore eu le temps de réapparaître.

Ajoutons aussi que nous avons toujours considéré le cas où le nombre de *spawns* est toujours supérieur ou égal au nombre de joueurs dans la partie.

7 Conclusion

Nous estimons que notre projet respecte toutes les spécifications demandées. De plus, nous avons ajouté quelques extensions à notre travail: deux bonus supplémentaires, un **GUI** modifié, un joueur supplémentaire et une petite série de tests.