UCLouvain

LINFO1131 - Project 2022
Capture the flag

Travail réalisé par Arthur Wery et Benoît Moedts

Groupe 59

NOMA : 60231900 | 09062000

Décembre 2022

1 Introduction

Dans le cadre de ce projet, il nous a été demandé de créer un jeu "Capture the flag" en OZ. Dans ce jeu, 2 équipes doivent ramener le drapeau de leur ennemi à leur base, l'équipe qui le ramène le plus vite à gagner la partie. Mais les joueurs peuvent se tirer dessus ou poser des mines pour ralentir ou éliminer l'ennemi. Nous avons dû créer le "Game Master" et des joueurs avec des comportements différents, l'interface GUI nous avait déjà été donnée. Pour réaliser ce projet, on ne pouvait utiliser que du code déclaratif.

2 Design de l'implémentation

Dans ce programme, 3 fichiers interagissent entre eux. En effet, comme on peut le voir avec la figure donnée dans les consignes 1, les joueurs interagissent seulement avec le "Game controller", qui lui interagit avec tout le monde, dont le GUI. Cette implémentation permet d'éviter toute triche, quand un joueur va vouloir faire quelque chose comme bouger, tirer, prendre le drapeau, il va demander au Game Controller qui s'il valide l'action (s'il n'y a pas de triche) va envoyer l'action au GUI qui changera l'interface. Nous allons voir l'implémentation de deux fichiers ou acteurs plus précisément. Le fichier Main.oz (le Game controller) et PlayerBasic.oz (Le joueur).

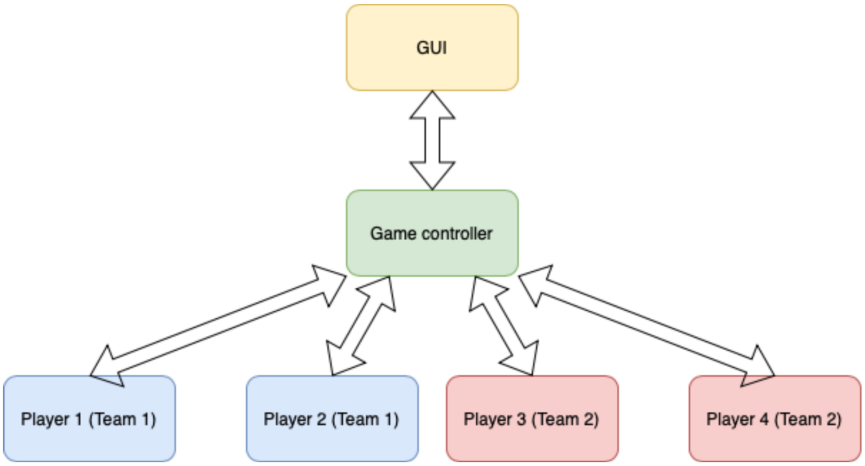

FIGURE 1 – Intéraction entre les acteurs

2.1 Main.oz

Ce fichier est le "GameController" du jeu, son rôle est de faire respecter les règles. Il va lancer le jeu et gérer les joueurs. Quand il lance le jeu, le main va créer un thread pour chaque joueur. Ce thread va gérer son joueur, il va demander s'il veut bouger, charger, tirer, prendre le drapeau, le déposer. Et s'il est mort, c'est ce thread qui attendra le temps du respawn, ce qui permet donc de ne pas bloquer les autres joueurs.
Chaque thread doit donc savoir où est son joueur pour savoir si ce qu'il fait n'est pas illégal. Cependant, avec un thread pour chaque joueur, il est impossible de savoir l'état actuel de la partie, car il n'y a pas de synchronisation. Nous avons donc créé un nouveau port object (GameStatePort). Ce dernier contient l'état de la partie et donc il va gérer la partie. En effet, après qu'un thread a

demandé à son joueur ce qu'il veut faire, il va envoyer ces informations au GameStatePort qui va vérifier si c'est dans les règles, auquel cas, il informera le GUI et les joueurs de l'action réalisée par le joueur. L'état de la partie comporte 4 variables, une liste avec la position des flags, une liste avec la position des mines, une liste avec la position de la nourriture, et une dernière liste avec l'état de chaque joueur. Pour chaque joueur, on stocke sa position, sa vie, son id, ses charges, et s'il a le drapeau ou pas. Ensuite pour générer la nourriture, au début du jeu, on lance un thread qui attend un temps aléatoire entre FoodDelayMin et FoodDelayMax quand le temps est écoulé, il va envoyer un message au GameStatePort qui s'occupera d'envoyer le message au GUI et de changer son état. Ensuite, le thread relance la fonction et ainsi de suite.
Pour résumer, chaque thread demande au joueur ce qu'il veut faire dans l'ordre qui nous a été demandé, ensuite, ils envoient au GameStatePort les informations qui va ensuite vérifier si c'est dans les règles et si oui, il va envoyer les informations aux joueurs et au GUI et ensuite changer son état pour finalement gérer les prochains messages.

2.2 PlayerBasic.oz

Ce fichier gère le comportement du joueur. Le player est un port object. Ce dernier n'envoie pas de message, il en reçoit uniquement. Il peut recevoir plusieurs messages comme move(), sayMoved(), chargeitem(), etc. Chaque message commençant par say est un message envoyé par le main qui donne une information sur un événement qui vient de se passer dans le jeu. Les autres messages comme move() ou chargeitem() sont des messages envoyés par le main pour savoir ce que veut faire le player. Le main donne en paramètre des variables unbound que le player va bound avec les valeurs qu'il souhaite. Ensuite, pour que notre player soit performant, il va enregistrer dans son état toutes les informations qu'il possède sur les autres joueurs. Le joueur stocke 14 informations sur la partie :

- id : Id du joueur.
- position : Position actuelle du joueur.
- hp : La vie du joueur.
- flags : La position actuelle des drapeaux.
- mineReloads : La charge pour la mine.
- gunReloads : La charge pour le fusil.
- startPosition : La position du spawn du joueur.
- mines : La position de toutes les mines.
- playersStatus : Une liste contenant l'état de chaque joueur, avec leur id, leur position, leur vie, s'ils ont le drapeau ou non, leur position de spawn, et la couleur de leur équipe.
- food : La position toutes les nourritures.
- hasflag : Contient le flag s'il le joueur en a un sinon contient false.
- path : Comporte une liste de positions pour aller d'un point A à un point B.
- teamColor : Comporte la couleur de l'équipe du joueur. On peut l'avoir avec l'id mais on trouve ça plus facile à comprendre comme ça.

En résumé, le joueur communique uniquement avec le main pour éviter toute triche et il n'envoie pas de messages, il affecte juste les valeurs données en argument dans les messages. Pour que le personnage puisse être intelligent, on stocke un maximum d'informations de la map et des joueurs adverses.

3 Stratégies des joueurs

Dans notre projet, nous avons implémenté 4 types de joueurs différents, les deux premiers sont les joueurs de bases, les deux derniers sont des joueurs améliorés. Pour tous nos joueurs, nous avons

implémenté un algorithme du plus court chemin (BFS).

3.1 Player059SimpleAttack

Déplacement : Il se déplace vers le drapeau ennemi jusqu'à ce qu'il puisse le prendre ou que quelqu'un prenne le drapeau ennemi, ensuite s'il a le drapeau, il va à la base avec, sinon il fait des mouvements aléatoires jusqu'à ce que le drapeau ennemi soit lâché. Une fois arrivé à sa base, il dépose le drapeau.
Armes : Il tire sur les joueurs ennemis qui sont à sa portée s'il y en a, s'il n'a pas de balle de fusil, il essaye de poser une mine.
Recharge : Il recharge son fusil s'il n'a plus de balle dedans, sinon il recharge la mine.

3.2 Player059SimpleDefence

Déplacement : Il se déplace vers le drapeau allié pour le défendre tout le temps, sauf quand un allié revient avec le drapeau ennemi, pour pas le bloquer, il fait des mouvements aléatoires.
Armes : Il pose des mines s'il peut, sinon il essaye de tirer sur un ennemi à sa portée.
Recharge : Il recharge la mine si elle n'est pas à 5 charges, sinon il recharge le fusil.

3.3 Player059OffensiveUpgraded

Déplacement : Il se déplace vers le drapeau ennemi jusqu'à ce qu'il puisse le prendre ou que quelqu'un prenne le drapeau ennemi, ensuite s'il a le drapeau, il va à la base avec, sinon il fait des mouvements aléatoires jusqu'à ce que le drapeau ennemi soit lâché. Une fois arrivé à sa base, il dépose le drapeau.
Armes : Il tire sur les joueurs ennemis qui sont à sa portée, il y en a, s'il n'a pas de balle de fusil, il essaye de poser une mine.
Recharge : Il recharge son fusil s'il n'a plus de balle dedans, sinon il recharge la mine. Mines :

3.4 Player059Tactical

Déplacement : Le chemin est initialisé à nil et est réinitialisé dès qu'un ennemi prend le drapeau, qu'un allié prend le drapeau ou que l'on respawn. Ce chemin est calculé en fonction de plusieurs facteurs : Est-ce que l'ennemi a le drapeau (enemyHasFlag), est-ce qu'un allié a le drapeau (allyHasFlag) et est-ce qu'on est en train de porter le drapeau.

On calcule le chemin selon cet algorithme :
Si un allié a le drapeau, on va aller essayer de se rapprocher de l'ennemi le plus proche du drapeau, si aucun ennemi n'est en vie, on ne bouge pas sauf si on bloque un allié, si un ennemi est en vie alors, on va vers lui sauf si ça bloque un allié. Dans le cas où on bloquerait un allié, on bouge dans une direction afin de débloquer cette situation.

Ensuite, si le chemin vaut nil ou qu'un ennemi a le drapeau, on calcule :
Si le joueur a le drapeau, il va à la base avec.
Sinon si un joueur ennemi a le drapeau et qu'on est à plus que 2 case de pouvoir prendre notre drapeau, alors on va vers l'ennemi qui a le drapeau, au contraire si on est proche de ramasser notre drapeau, on va quand même le ramasser. Sinon, on va chercher notre drapeau le plus vite possible.

Armes : S'il a assez de charges pour une mine et qu'il a le drapeau ennemi, alors il pose une

mine, sinon s'il a une balle pour tirer, il essaye de tirer d'abord sur une mine sur son chemin s'il y en a une et qu'il n'y a pas d'allié sur celle-ci (à 1 ou 2 de distance), sinon il essaye de tirer sur un joueur ennemi à sa portée.
Recharge : Il recharge son fusil s'il n'a plus de balle dedans, sinon il recharge la mine.

4 Bonus

Nous avons implémenté 3 bonus, des joueurs supplémentaires, la réaction chainée des mines et la génération aléatoire de map.

4.1 Les joueurs supplémentaires

On a créé d'autres joueurs avec des comportements différents, tout est expliqué dans la section "Stratégies des joueurs"3

4.2 La réaction chainée des mines

Quand une mine est déclenchée par un joueur qui marche ou qui tire dessus, si une mine est à 1 de distance de Manhattan de la mine qui vient d'être déclenchée, alors elle explose aussi. Tout ceci a été implémenté récursivement.

4.3 Génération aléatoire de map

Dans Input.oz, on génère une nouvelle map. Tout d'abord, on génère une matrice de 12x12 avec des variables unbound en utilisant List.make. Ensuite, on génère un chiffre entre 0 et 1. Si c'est 0 les deux équipes seront situées sur la première et la dernière rangée. Sinon, elles sont situées sur la première et dernière colonne. Après, on génère un chiffre entre 0 et 10. Ce chiffre sera la première case de la base de l'équipe 1. On génère donc les bases des deux équipes avec cette information, car la base de l'équipe 2 est à l'opposé à celle de l'équipe 1. Ensuite, on bind les tiles en face et au milieu des 2 bases pour avoir une place pour le drapeau. Finalement, on parcourt toute la matrice, et à chaque fois, on génère un chiffre entre 0 et 4 et si le chiffre vaut 0 on met un mur et donc 3 dans la matrice, sinon on met 0. À la fin de la création de la matrice, on regarde s'il y a un chemin grâce à la fonction ShortestPath qui est basée sur l'algorithme BFS. S'il n'y a pas de chemin, on recommence, sinon on renvoie la map, les drapeaux et les positions des spwans.

5 Interopérabilité

Dans les tests, on fait à chaque fois 6 matchs en inversant les côtés après trois matchs

Player059Tactical vs Player018Tactical :
Le player adverse a l'air de trouver le bon chemin mais n'attaque pas assez. Pas d'amélioration par rapport a ce player.

Player059Tactical vs Player006TankAttacker :
Ce player attaque un peu plus mais il a du mal a trouver le chemin vers le drapeau et ne pose pas de mine. Le scénario le plus proche de notre défaite était celui ou les ennemis tiraient avant nous (Peaker Advantage).

Player059Tactical vs Player120Berserker :
Le player ennemi a un pathfinding douteux, en conséquence la victoire a été facile. Même avec une génération aléatoire très désavantageuse, notre player gagne.

Malheureusement nous n'avons pas pu tester avec des bots très compétitifs, nous aurions aimé affronter des bots qui battent le nôtre afin de l'améliorer et de trouver encore plus d'amélioration.

6 Conclusion

Ce projet "Capture the flag" nous a permis de comprendre parfaitement les ports objects et leur utilité, le plus gros problème que nous avons rencontré est la synchronisation de l'état du main. En effet, comme expliqué précédemment, comme le main utilise un thread par joueur, il fallait trouver un moyen pour que chaque thread possède le même état. Nous avons pu grâce aux ports objects régler ce souci. De plus, dans ce projet, nous avons pu nous surpasser en inventant un bon joueur qui puisse battre les autres joueurs. Nous tenons à vous remercier pour ce projet qui nous a beaucoup appris et amusé en même temps.