LINFO1131 - Project 2022
Capture the flag

1 Introduction

For this project you are asked to implement a customized version of a popular game, capture the flag. In capture the flag, two teams are facing each other trying to bring the enemy flag back to their base. Players can slow each other down by placing mines and shooting each other. The game will be played simultaneously, not turn by turn.

You will design the behaviour of the players, deciding where they should move to, if they should charge a weapon, use a weapon, and so on and you will implement the rules of the game as well. Players will have to communicate with the game master to make sure they are not breaking any rules. Informations on the battlefield (mine positions, player positions, flag positions) are known to players from both teams as long as they are alive.

2 Rules of the game

This section will cover the basic rules of the game that your implementation should respect.

- Players start with 2 lives. Each team is composed of 3 players.
- Players can move by one tile vertically or horizontally. Walls and the enemy base are impenetrable : players can't move there. If two players want to move to a tile at the same time, the first one to ask gets to move there and the other one cannot move this turn.
- If a player is dead, it has to wait `RespawnDelay` before it can respawn. It then respawns with `startHp`.
- Players have two weapons they can use : a gun and a mine. Each item must be charged before it is used : charging the gun requires 1 action, charging the mine requires 5.
- Mines are placed under the player that is placing it, and stepping on a mine will set it off, giving 2 damage to the player on that tile and 1 damage to the players that are 1 tile away (following Manhattan distance).
- The gun has a range of two (Manhattan distance), and targets include players as well as mines. Shooting a player will cause 1 damage, and shooting a mine will make it explode.
- In order to win the game, one of the players has to pick-up the enemy flag (the player has to be standing on the flag to grab it), bring it back to its base and drop it there. If a player dies carrying the flag, it is dropped at the player's position. Players can't pick-up their own flag.
- Food will appear randomly on the map. A player can consume it by standing on it, and that will add 1 to their life.

3 Program description

This section describes the expected architecture of your program and the motivations behind it.

3.1 Requirements for the automated tests

Your project will be partially automatically tested. This requires you to:

- Respect the file structure and the message format that is detailed in the following sections;
- Remove any call to the Oz `Browser.browse`, use `System.show` instead.

Following these guidelines is mandatory to make the grading scripts work. The next subsection will describe the different components of the program.

3.2 Types

To ease modularity, we define some types to respect. These are defined using EBNF grammar.

```
<idNum>          ::= 1 | 2 | .. | Input.nbPlayers
<colorNum>       ::= 0 | 1 | ... | 255
<color>          ::= red | blue | green | yellow | white | black | c(<colorNum> <colorNum> <colorNum>)
<row>            ::= 1 | 2 | ... | Input.nRow
<column>         ::= 1 | 2 | ... | Input.nColumn
<weapon>         ::= gun | mine | null

<id>             ::= null | id(name:name color:<color> id:<idNum>)
<position>       ::= pt(x:<row> y:<column>)
<firedWeapon>    ::= <weapon>(pos:<position>)
<flag>           ::= flag(pos:<position> color:<color>)
<food>           ::= food(pos:<position>)
```

3.3 Parameters of the game (`Input.oz`)

The game has many parameters, leading to many different games. Following are the parameters and their descriptions (the `Input.oz` file contains the setup parameters of the game) :

- Description of the map :
  - `NRow` : The number or rows of the map;
  - `NColumn` : The number of columns of the map;
  - `Map` : The description of the map, 0 meaning there is nothing, 1 being the first team's base, 2 being the second team's base and 3 being walls. For example, the following map description gives a map with 6 rows and 12 columns with the first team's base on the top right and the second team's base on the bottom left :

```
Map = [[1 1 1 0 0 0 0 0 0 0 0 0]
[1 1 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 2 2 2]
[0 0 0 0 0 0 0 0 0 2 2 2]]
```

- Description of the players :
  - `NbPlayers` : The number of players
  - `Players` : A list of the players that will play (example : `Players = [player1 player2 player1 player2]`) where `player1` and `player2` are different implementations of players
  - `Colors` : The corresponding colors (teams)

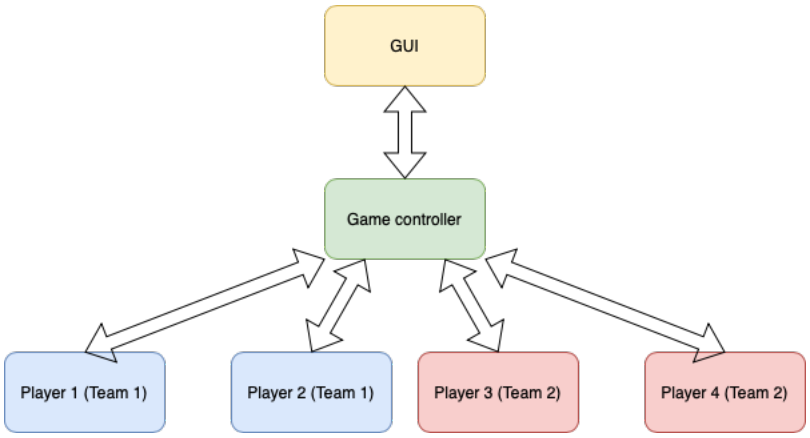

Figure 1: Interaction between the different modules

  - `SpawnPoints` The positions where players can spawn
  - `StartHealth` : The initial health of the players
  - `ThinkMin and ThinkMax` : The bounds of the random simulated thinking time
  - `GunCharge and MineCharge` : The number of times you have to charge a weapon for it to be useable
  - `RespawnDelay` : The amount of time (ms) a player has to wait before it can respawn.
- `FoodDelayMin and FoodDelayMax` : The bounds of the time between two food spawns.
- `GUIDelay` : The delay between GUI effects
- `Flags` : The initial positions of the flags for both teams

Guidelines for modularity : Don't change any names, only change the values :)

3.4 Game controller (`Main.oz`)

The game controller is the master of the game. It makes sure that the players only perform actions that they are allowed to perform. It also coordinates the different players and their interactions, and sends the information to the GUI, so that other people can watch the game. It is in charge of creating the GUI Port as well as the ports for all players.

Players should only interact with the game controller, and not with each other. Similarly, only the game controller should interact with the GUI, not the players (figure 1).
You are provided with a basic version of the game controller that creates the port object for the GUI and launches its interface.

At the start of the game, it should do the following (in this section, "notify everyone" means the players and the GUI):

- Create the port for every player and assign each of them a unique id between 1 and `nbPlayer` (type $< idnum >$). The ids are given in the order they are defined in the input file;
- Set up every player (select a type of player, create a thread for each player, spawn them on the map);
- When every player has set up, launch the game.

Let's now explain how the game works. Each "turn" (the sequence of operations is repeated and can be viewed as a turn even though all players are playing simultaneously) is composed of the following sequence of operations :

1. If the player is dead, the main waits for `RespawnDelay` and then sends `respawn()` to that player so it knows it can set its local state's hp to `startHp` and start playing again. Update the main's state for that player to set its health to `startHp` and then proceed with the rest of these steps.
2. If the player is alive, ask it where it wants to go.
3. Check if the position the player wants to move to is a valid move. If it is not, the position stays the same (the player cannot move). Otherwise, notify everyone of the player's new position.
4. Check if the player has moved on a mine. If so, apply the damage and notify everyone that the mine has exploded and notify everyone for each player that took damage. If a player dies as a result, notify everyone and skip the rest of the "turn" for that player.
5. Ask the player what weapon it wants to charge (gun or mine).
6. Ask the player what weapon it wants to use (place a mine or shoot at something). Check if the player can indeed use that weapon, and if so send a message notifying everyone, then reset the charge counter to 0 for that weapon. If a mine is exploded as a result, notify everyone that it has exploded and apply the damage. If a player has been shot, notify everyone.
7. Ask the player if it wants to grab the flag (only if it is possible). Notify everyone if the flag has been picked up.
8. If applicable, ask the player if they want to drop the flag. Notify everyone if they do.
9. If a player has died, notify everyone and also notify them if the flag has been dropped as a result.
10. The game Controller is also responsible for spawning food randomly on the map after a random time between `FoodDelayMin` and `FoodDelayMin` has passed.

Remember that the game controller has to make sure all players respect the rules, so it has to keep track of things like health, weapon charges,... for each player.

3.5 The graphical interface (`GUI.oz`)

This functor has only one export (`portWindow`) linked to a function creating a port object and launching in a thread the treatment of its stream.

The port object can handle the following messages (coming from the game Controller):

- `buildWindow()` : Create the window
- `initSoldier(ID Position)` : Add player `ID` to the game at position `Position`. `ID` is of type $< id >$ and `Position` is of type $< position >$.
- `moveSoldier(ID Position)` : Move the player `ID` to position `Position`
- `lifeUpdate(ID Life)` : Update player `ID`'s health to `Life`
- `putMine(Mine)` : Place the mine (`Mine` is of type $< firedWeapon >$)
- `removeMine(Mine)` : Remove the mine
- `putFlag(Flag)` : Add the flag (`Flag` is of type $< flag >$)
- `removeFlag(Flag)` : Remove the flag
- `putFood(Food)` : Add the food (`Food` is of type $< food >$)
- `removeFood(Food)`: Remove the food
- `removeSoldier(ID)` : Remove player `Id` from the game

You are provided with a working GUI that can display the basic elements of the game. You can add to it (see bonus section) if you want to add features.

Guidelines for modularity : Don't add any export, don't import other files from the project, program in a defensive way. The GUI is only receiving information, it is just used to display the state of the game.

Consult the following resources for designing GUI:

- Book chapter 10 (page 679-705)
- `https://mozart.github.io/mozart-v1/doc-1.4.0/mozart-stdlib/wp/qtk/html/index.html`

3.6 Players (`PlayerXXXMyCustomName.oz`)

Each player is, as the GUI, summarised by a port object. The basic structure will thus be close to the one of the GUI.

You are asked to design 2 basic behaviours for the players, that you will then be able to customize. Both versions should be able to handle all the messages specified. Starting with a random one is a good idea to get used to the rules.

Question marks are only there to ease the reading, meaning the parameter concerned is initially unbound and the initialisation is done during the treatment of the message (the player has to do the binding). To simulate thinking, the player should wait a random time between `ThinkMin` and `ThinkMax` when it has to bind parameters to make a decision.

The basic player should be able to handle the following messages (coming from the main) :

- `initPosition(?ID ?Position)` : Allows to set the players spawn position. `ID` is of type $< id >$ and `Position` of type $< position >$.
- `move(?ID ?Position)` : Asks the player where it wants to move. `ID` is of type $< id >$ and `Position` of type $< position >$.
- `sayMoved(ID Position)`: Tells the player that player `ID` is now at position `Position`; `ID` is of type $< id >$ and `Position` is of type $< position >$.
- `sayMineExplode(Mine)`: Tells the player that the mine has exploded (mine is of type $< firedWeapon >$).
- `sayFoodEaten(ID Food)`: Tells the player that player `ID` has eaten the food. `ID` is of type $< id >$ and `Food` is of type $< food >$).
- `chargeItem(?ID ?Kind)`: Allows the player to choose a weapon to charge. `ID` is of type $< id >$ and `Kind` is of type $< weapon >$.
- `sayCharge(Id Kind)`: Tells the player that his weapon Kind has been charged. `ID` is of type $< id >$ and `Kind` is of type $< weapon >$.
- `fireItem (?ID ?Kind)`: Allows the player to choose a weapon to fire. `ID` is of type $< id >$ and `Kind` is of type $< firedWeapon >$.
- `sayMinePlaced(ID Mine)`: Tells the player that a mine has been placed by player `ID`. `ID` is of type $< id >$ and `Mine` is of type $< firedWeapon >$.
- `sayShoot(ID Position)`: Tells the player a gun has been fired towards position `Position` by player `ID`. `ID` is of type $< id >$ and `Position` is of type $< position >$.
- `sayDeath(ID)`: Tells the player that player `ID` is dead. `ID` is of type $< id >$.
- `sayDamageTaken(ID DamageTaken LifeLeft)`: Tells the player that player `ID` has taken damage and how much life it has left. `ID` is of type $< id >$ and `DamageTaken` and `LifeLeft` are integers.
- `SayFoodAppeared(Food)`: Tells the player that food has appeared on the map. **Food** is of type $< food >$.

- `takeFlag(?ID ?Flag)`: Ask the player if it wants to to take the flag or not. `ID` is of type $< id >$ and `Flag` is of type $< flag >$.
- `dropFlag(?ID ?Flag)`: Ask the player if it wants to drop the flag. `ID` is of type $< id >$ and `Flag` is of type $< flag >$.
- `sayFlagTaken(ID Flag)`: Tells the player that player `ID` has picked up the flag `Flag`. `ID` is of type $< id >$ and `Flag` is of type $< flag >$.
- `sayFlagDropped(ID Flag)`: Tells the player that t`Flag` has been dropped by player `ID`. `ID` is of type $< id >$ and `Flag` is of type $< flag >$.

Guidelines for modularity : We ask you to name your player following this rule : The file is in the form `PlayerXXXMyCustomName.oz`, `XXX` being replaced by your group number (004, 056, 103,...) and `MyCustomName` by any descriptor you can chose to differentiate your different players. The name (used in the input file) associated with your player file will be the name of your file, in lowercase. For example, `Player042BasicAI.oz` contains the player named `player042basicai`. This will ensure every player made by each group has a different name.

3.7 Selection of the right player (`PlayerManager.oz`)

This file is responsible for facilitating the selection of the player following the type of the players given in the input file.

This file shall be the only one (with the input one to change the names) to modify to make different players playing together. To add a new player, the name of the file should be added to the import section and a pattern matching line should be added to create the corresponding port object for a player.

Guidelines for modularity : Nothing should be done here except adding players to the list.

4 Functors and compilation

For syntax and use of functors, consult book pages 220–230.

Compiling functors : `ozc -c myfunctor.oz`

Executing functors : `ozengine myfunctor.ozf`

You are provided with a `Makefile` that runs the empty project you are given. When typing `make all` in the command line, it will create the GUI window and display the initial state of the game. However, nothing else will happen as it is your job to implement it. You have to change the `Player1` and `Player2` to compile the corresponding files you will have implemented, these are just here so you have a working project to start with. Keep in mind the guidelines for how to name the players. You can do the project on any OS you like, but the project should compile and run on the computers in the Intel room. If you plan on working with Windows, the Makefile will not work so you have to compile the project yourself.

4.1 On mac Os

The `Makefile` should work for macOs as well, but if it doesn't here is how to fix it. You can find the path to ozc and ozengine by going to your application folder, "click with two fingers" (right-click) on Mozart, chose the second item in the menu (something about seeing the content), the folder of the application will open and you will be able to find the bin folder somewhere containing all the oz binaries. By going to the properties of the ozc or ozengine, you will be able to get their full path and you can update the path in the `Makefile`.

5 What do you have to do (summary)

This work has to be done by groups of 2 people (not alone!). The deadline is **December the 14th at 11:59pm**. Your code should only use threads, instructions to create ports and send messages for the communication between the game controller, the Players and the GUI. Your code should not contain explicit

cells, but only declarative code and possibly (The whole project is feasible only with declarative code) active objects for the internal structure of your elements.

5.1 Submission

In order to submit your project, you must be registered in a group on INGInious.

All of the following files must be submitted in the following task in a `.zip` archive. It should at least contain:

- A `GUI.oz` file for the GUI;
- A `Main.oz` file for the game controller;
- Two different players (the first two should have strictly different behaviours for everything). You can have more than two players.
- A `Input.oz` file for the input of the game.
- A `PlayerManager.oz` file to manage the different types of players.
- A `Makefile` to run the project (it should be the one we gave you with small modifications)..

Additionally, you are expected to write a report explaining your work. This report must not exceed 5 pages (all included) and must at least detail the following things:

- Each of the player strategies;
- The design of your implementation and the choices that you made (don't copy/paste your code, explain the reasoning behind it);
- Any extension (see next section) that you implemented;
- Results of the interoperability, and possibly how you improved your players based on that (see section 6).

5.2 Extension Part (how to get more points)

Implementing all the features described above will result in a maximum grade of **14/20**. To get more points, you can develop additional features. This is a list of ideas to improve the project; you don't have to do them all to gain a bonus and you don't have to do them in order. Feel free to follow your own ideas (you can post them on the forum, so we can discuss about them)!

- Create a human player file, creating its own window making it possible for a human player to answer the messages sent by the game controller (in that case, don't forget to update the simulated thinking time to make it realistic) : **4 points**
- Create a generator of maps in the Input file, creating the map used: **1 point**
- Create other players files, with other behaviours and strategies. There can be overlap of strategies in these files (same displacement but different firing approach,...). Make sure to explain the strategies you implemented(e.g. more defensive players, more offensive players,...); **up to 4 points depending on how complex the behaviour is**
- Add sound effects and visual effects to the GUI for when mines explode, when guns are fired, ... **2 points**
- Add a new tactical weapon, "air strike", that can be unlocked and used once by going to a specific tile on the map called "Mercenaries". It will do area damage similar to the mine, but dealing 3 damage where it hits and decreasing the damage by 1 according to the Manhattan distance (implement the mercenaries tile like the food mechanic). It can hit anywhere on the map; **2 points**

- Spawn bonuses randomly on the map for players to pick up similarly to how food works: speed +1 for 5 seconds and health + 2 for 5 seconds (you can add more than these two); **2 points**
- Allow mines to explode other mines if they are adjacent (chain reaction). **1 point**

Your extensions must be detailed in your report to be fully evaluated. You may add up to 2 pages explaining your extensions, for a total of 7 pages for the whole document, figures included.

6 Interoperability

If you follow the specifications, you will be able to test your implementation with players of other groups (if you add features to the battlefield, some players might not be able to interact with them so keep that in mind when your are testing). We ask you to test your player against at least 3 different players from 3 different groups (the basic random players excluded). Feel free to test with more.

When sharing your player with another group, you can't share the code! You have to compile it on your side and give only the `.ozf` compiled file to the other group so they won't access your code (as sharing code is forbidden). The list of players (and groups) you have tested should be given in the report with some remarks on how it went, if it helped you find mistakes,...

7 Q & A

7.1 General

- **Can we consider the input file as correct? Do we have to check the values?** The input file is considered as right. No need to check the type of the variables.
- **Can we put utility functions in another file for the methods used by all the players?** No, your players should be self-contained for the interoperability sessions.
- **How do we read the position?** The $(1, 1)$ point is at the left top corner. The squares positions are read as matrix cells. $(1, 7)$ is the square at the 1st line, 7th column. $(5, 2)$ is the square at the 5th line, 2nd column.
- **Can we discuss about the implementation of the project with other students?** You are allowed to discuss with other groups about the project, but you are not allowed to share code between groups.
- **If we have** 100% **on** `INGInious`**, does it mean We have** 100% **for the project?** No, INGInious only makes sure that your submission has the right format.