LINGI1131 — Project Report: *Captain Sonoz*

Hadrien Libioulle (10-04-1700) Louis Laurent (68-06-1700)

Wednesday, 29th April 2020

1 Introduction

This report has been created for the project "Captain Sonoz" for the course LINGI1131. The aim of the project was to implement a customized version of the game "Captain Sonar" using the oz language.
In this report can be found three main sections: firstly, our design choices along a view of the implementation, then an overview of our players strategies and finally a description of our extensions.

2 Player Strategies

1. **Player064Dumbest:** This player is pretty basic. In fact, to choose its initial position, it selects a random free position on the map. In terms of movements, it is a pseudo random player moving in loops. It tries to move North, then East if North is impossible, then South if East is impossible, then West if South is impossible and finally, it surfaces if all directions are impossible.

   When asked to load/fire an item, it loads/fires[1] in the following order:

   missile → mine → sonar → drone

   When asked to fire a mine, it fires the last placed mine.

2. **Player064Overkill[2]:** Everything is the same as **Player064Dumbest** except :

   - Movements
   - Strategy to fire items (which is based on a strategy to find the other players)

[1] only if the corresponding item has enough charges
[2] aka. *Jadinator*

Movements of this player are even more random,[3] to sweep the map even more. The strategy to fire items is more complex. In fact, to find the other players, this player is listening to the messages **sayMove(ID Direction)** to trace the others and incrementally, remove possible initial positions for each opponent. Once there are not many possible initial positions, the *overkill* player has found his opponents and he can kill them as soon as they are in reach.

3 Implementation & design choices

The main procedure:

- creates a list containing all the player ports.
- builds the GUI window.
- places each player at their chosen position.

Then, depending on the **IsTurnByTurn** variable of the **Input** class, it does one of the following:

3.1 Game Controller - Turn by Turn

The procedure **GameTurnByTurn** is called. This procedure is recursive and takes 4 arguments:

- **Step**: Corresponds to the different steps a player has to go trough between the beginning and the end of his turn (i.e. diving, moving, loading an item, firing an item, detonating a mine and ending the turn).
- **CurrentId**: The index of the corresponding player port in the **PlayerList** variable.
- **Surface**: A list with as many elements as players. The value ranges between -1 and **NbTurnSurface**-1 (since the first turn at surface counts). A positive values indicates that the submarine is at the surface and more precisely the number of turn(s) remaining before the submarine is allowed to dive again. 0 means the submarine is allowed to dive (i.e. the submarine finished waiting or it is his first turn). -1 means the submarine is currently underwater.
- **PlayerDeadList**: A list with as many elements as players. For each player the value is either 0 (alive) or -1 (dead). This allows the game controller to loop only over the alive submarines (thanks to the **NextId** function) and determine when the game is over.

Together, these variables constitute a state passed from an execution to the next and allows the game controller to follow the state of the game. We begin with the first player at the first step, then the step increments until the first player's turn is over. The second player follows the same path and so on.

[3] random choice between the five directions

3.2 Game Controller - Simultaneous

In this game mode, we create a thread for each player as well as a port (**SimulPort**) object and stream (**SimulStream**) object that allow all the thread to be connected to a game controller. Each player loops over the recursive procedure **SimulGaming** very similar to **GameTurnByTurn**, only this one does not have the **PlayerDeadList** as an argument and the **Surface** variable is not a list but just a number.
After all the thread have been launched, the main thread launches the recursive procedure **GameSimultaneous** that treat the created **SimulStream**. The main purpose of this procedure is to handle the **PlayerDeadList** (i.e. allowing all the threads to have access to the information, and tracking when the game is over).

3.3 Explosion and Messages

When a missile is detonated or a mine detonated, the function **ExplosionHandling** is called. It is a recursive function that asks every player if he was hurt by the explosion. The function **MessageHandling** is then called, it uses the information queried by **ExplosionHandling** to update the life or remove a player in the GUI. In this function, and when a message needs to be broadcast to all players in the functions **GameTurnByTurn** or **SimulGaming**, the procedure **BroadcastMessage** is called.

3.4 Players

- To avoid previously visited tiles, both custom players are using a "private" map where each visited tile is replaced by a 1 (so they are considering these tiles as islands) and when they try to move, they use this map instead of the map in the input file.
- Apart from that, I'll mainly focus on the *overkill* player and its strategy to find and kill his opponents because the other functions are relatively straightforward when looking at the explanations of strategies. I'll divide the explanation in two parts : find strategy and fire strategy.

  1. Find strategy — mainly based on the following functions :
     - **{UpdateDirections ID Directions Others}**
     - **{OtherIsDead ID Others}**

     So, it works as following :
     - Search in ***Others***[4] the player corresponding to ***ID***
     - When found, it updates the map corresponding to the player by going through each line and column, searching for all positions where the player cannot be (i.e. islands and outside the map). Then it follows the player's path in the opposite way until the player "hypothetical" initial position is reached. This starting position is impossible because this player would have been on an island or outside the map at the game start. The position is thus marked as impossible (by a -1 on his map).
     - It returns the new map of possible initial positions and the update number of possible initial positions.

[4] This is a list of records containing for each opponent: if he is alive or not, his path, his map containing all possible initial positions and the current number of possible initial positions

2. Fire strategy — mainly based on the following functions :
   - ***{SelectPlayer NbStartingPos Others Acc}***
   - ***{AllPossibleInitPositions Other Min Max}***
   - ***{Loop N Type}***
   - ***{FindHitPosition OppPosition MinRange MaxRange}***

   So, it works as following :
   - The "*overkill*" player chooses the first loaded item in his list of loading items.
   - Then, he searches for every possible initial positions[5] for all opponents having less than the number of map's tiles divided by 5 possible starting positions[6].
   - If an opponent is within range then it finds the nearest hitting position[7]. By that, we mean the nearest position from the opponent that is still within reach by a missile/mine.

3.4.1 Sonar & Drone

We are not using them in both players we have made for the following reasons:

- They are both relatively random in their result.
- The efficiency of both is related to the number of charges needed which is difficult to implement[8] or inefficient[9].
- Our implementation of the finding strategy only depends on the size of the map, its configuration and the movements of all the opponents.

So, we have concluded that implementing an efficient finding strategy based on sonars & drones would be very complex with probably little to no result.

4 Extensions

4.1 Multiple player strategies

We have made 2 players with 2 different strategies : a pseudo random player[10] and a more intelligent player[11]. These players and their strategies have been described in the "Player Strategies" section and in the "Implementation" section.

[5] using the function ***{AllPossibleInitPositions Other Min Max}***
[6] using the function ***{Loop N Type}***
[7] using the function ***{FindHitPosition OppPosition MinRange MaxRange}***
[8] because we should have done a player based on the input file with multiple strategies
[9] if we made a basic player that would have based his finding strategy on sonars and drones, his strategy would have been inefficient with high charges number
[10] aka. ***Player064Dumbest***
[11] aka. ***Player064Overkill***

4.2 Animations & Images

1. Explosion: the image of a blast is displayed from the center of the explosion toward the sides.
2. Drone: a line of white squares with an image of a drone crosses the line it is asked to check.
3. Sonar: a text message is displayed on the header.
4. Mine : mines are represented using the color of the player who placed it and an image of a mine.

4.3 Map Generator

4.3.1 How to use it?

1. You have to manually set the variables ***UseMapGenerator*** in the ***GUI.oz*** and in the ***Input.oz*** to *true* in order to enable this extension.
2. If the extension is enabled, you will have to click on the ***Regen*** button in the top right corner of the window until you see the map you want (the process to generate the maps is random).
3. When you have chosen a map, you can click on the ***Ok*** button to launch the game with the corresponding map.

⚠ Warning ⚠ : if this extension is disabled, the ***NRow*** and ***NColumn*** variables in the ***Input.oz*** file must match the **DefaultMap** in order to work flawlessly.

4.3.2 How does it work?

If the extension is *enabled*, the *GUI* will display two buttons in the toolbar with a corresponding *Port Object* in order to treat signals[12] coming from them:

- When the ***Regen*** button is clicked, a *regen* signal is sent to the ***GeneratorPort*** which is managed by the procedure ***{GenerateMap S G M}***. This signal causes the regeneration of the map using the function ***{MapGenerator}*** and the redrawing of the map using the function ***{DrawMap Grid Map}***.
- When the ***Ok*** button is clicked, an ***ok*** signal is sent to the ***GeneratorPort***. It causes the current map to be saved in the ***Input.map*** variable in order to be visible from everyone.

[12] which are the ***regen*** and ***ok*** signals.