LINGI1131 - Project 2019
Ozploding bozmbs

Hélène Verhaeghe & Guillaume Maudoux

07-04-2019 (version 1)

1 Introduction

This document presents the specification of the project of LINGI1131. Please, read this carefully before beginning anything.

2 Summary of what is asked

For the **3rd of May 2019, 18:00** (the deadline is strict), by **groups of maximum two**, you have to submit on **INGInious** a **zip** containing:

- a correct **makefile** providing all the required commands (see Section 5.2) with the completed header
- the player files (see Section 7.6)
- the player manager file (see Section 7.5)
- the GUI file (see Section 7.4)
- the main file (see Section 7.3)
- the input(s) file(s) (see Section 7.2)
- the compiled functors against which you performed you interoperability tests (see Section 5.3)
- additional files that may be required by your project (common functors, text files, . . .)
- a report of **maximum 5 pages** containing all the required elements (see Section 6)

The **mandatory part** of the project requires:

- a working turn by turn controller
- a working simultaneous controller
- a working basic player
- a working more advanced player
- an interoperability test with the provided `Player000bomber.ozf`
- 5 more interoperability tests with distinct players from at least 3 differents groups

The **optional but somewhat required part** of the project must contain extensions to the above requirement, possibly including (but not limited to):

- tuning the GUI to your liking
- implementing smarter, more advanced players, keyboard controlled players, etc
- doing more interoperability tests, or an automated tool for interoperability testing
- develop a strong test suite for you application
- initiate and lead a sharing platform for compiled agents
- add some other kind of features or tooling around this project

Fulfilling the mandatory part of the project is worth 14/20. Adding elements from the optional part will bring you more points but won't compensate for a broken mandatory part. A mandatory demo session of 10 min will be organized after the deadline. You will be asked to showcase your work on some scenarios and answer live questions. The demo will be graded in a bonus/malus fashion (-2, -1, 0, 1 or 2) based on the quality of the demo and on your answers. Not showing to the demo will automatically result in a -2.

Questions have to be asked on the forum section of Moodle.

3 Hints

If you don't know how to start, you can follow this order (but this is not mandatory to follow):

1. Read this file carefully
2. Implement the initialization part (display window, add player, spawn player, make them do one move,...)
3. Create your makefile and already write the basic rules to compile your project
4. Implement the turn by turn part (using the given GUI and the given player)
5. Implement your own purely random player and complete your makefile
6. Implement the simultaneous part
7. Create your better player
8. Do some interop and correct the bugs
9. Write the basis of your report
10. Submit a first time on Inginious
11. Improve your code and implement some optional task
12. Finish what's left
13. Submit your final code

This list is not complete, not mandatory to follow but may help you structure your work if need be. We insist on taking small steps and testing often.

4 Rules of the game

Your task is to implement in Oz a version of the well known Bomberman game. In Bomberman, the players (named bombers), are on a 2D grid board. This board is composed of floor tiles, wall tiles and box tiles. Obviously, the bombers can only move on floor tiles. They can move in the cardinal direction (north, south, east, west). The bombers have one ability: they can put bombs on the floor. When exploding, the bombs trigger some fire that propagate in the cardinal direction (north, south, east, west). Fire propagation is done following theses rules:

- fire propagates towards the four cardinal directions starting from the exploding bomb
- fire is blocked by wall tiles
- fire on a box tile destroys the box and stoppropagating
- a player standing in the way of propagating fire (in the same tile) gets killed, even when its own bomb triggered the fire
- fire stop propagating if it goes too far from its source (the bomb that trigger it)

A destroyed box release one point if it is a point box or a bonus and unveil a floor tile. The basic bonuses possible is either a new bomb, either 10 points (50/50 possibility). Other bonuses are considered as extensions. The goal is to be the bomber with the more point at the end of the game. Three ends are possible:

- there is only one bomber standing (still alive)

- there are no boxes left
- there are no more bombers standing (all dead)

Some border cases: two bombers can be on the same tile at the same time, two bombs from two different bombers can also be on the same tile at the same time but two bombs from the same bomber can not be on the same tile at the same time.

4.1 Game modes

You are asked to implement two different game modes: "turn by turn" and "simultaneous".

Turn by turn In turn by turn, the first part is to determine the order (at random) between all the players, to assign them spawn position and to initialise the GUI.

Then begin the main part which loops over the players following the order defined. Each turn of a player consists of doing an action (move or bomb) if it is possible (if player is on the board, if not, just skip its turn).

Simultaneous In simultaneous, the initialisation is first done (spawns assigned, initialisation of GUI).

Then each player is launched simultaneously. To avoid a pseudo turn by turn due to operation taking more or less the same time for each, a thinking delay (delay random between some bounds given in `Input.oz`) has to be inserted by the agents before emitting a significant `doaction` message.

5 Compilation of the project and testing

5.1 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`

To make the whole project working, first compile the Input.oz file, PlayerManager.oz file, players files, GUI.oz and Main.oz. Then execute the created functor file Main.ozf.

Remark for Mac OS users : You can find the path to ozc and ozengine by going to your application folder, "click with two fingers" 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 execute them by command lines.

5.2 Makefile

You are asked to provide us a correct `makefile`[1] with the following rules:

- `all`: compile all your files and launch the game
- `compile`: compile all your files (except the one concerning the inputs)
- `compilePlayers`: compile the `PlayerXXXname.oz` file(s) to create the corresponding `PlayerXXXname.ozf` (and, if you have some, other files that are part of the player module)
- `*.ozf`: compile the corresponding `.oz` file to create the corresponding `.ozf`
- `run`: launch the game
- `clean`: remove all generated file (`*.ozf` files)

[1]Go back to your Operating System course or look at the internet (`https://www.gnu.org/software/make/manual/html_node/Simple-Makefile.html`) to remember how to make a correct makefile ;)

5.3 Interoperability tests

You are asked to exchange your PlayerXXXname.ozf (the *.ozf file and not the *.oz as no code can be shared) with other groups. You are asked to test your main module with at least 5 players from at least 3 differents groups (notify in the report which player from which groups you tested and submit their PlayerXXXname.ozf in your archive). This will help you test your code and find bugs or misunderstanding.

Remark: Think about setting the UseExtension (see Section 7.2) boolean to false when doing interoperability tests with players who doesn't implement your extensions.

6 Report

You are asked to provide us a maximum 5 pages long report (cover page and table of content not included in the 5 pages). Your rapport should be well structured (introduction, conclusion, sections,...), mention your name(s), surname(s), noma(s) and group number and shouldn't repeat what was explain in this file (we know what you were ask to do in detail). The introduction should mention which mendatory tasks you didn't manage to do and which optionnal tasks you added to your project.

Your repport should contain explanaitions on the structure of your implementation for the main module, explanaitions on your players, list the groups and their player you interop with, explainations on the results of the interop tests (averything went well, you found some bugs,...), explanaitions on your extensions, details about what problem you faced,...

7 Specifications

7.1 Types used (EBNF formula)

The EBNF formulations of the data structures used are the followings:

```
<bomber>      ::= bomber(id:<idNumP> color:<color> name:<name>)
<idNumP>      ::= 1 | 2 | ... | Input.nbPacman
<color>       ::= red | blue | green | yellow | white | black
                  | c(<colorNum> <colorNum> <colorNum>)
<colorNum>    ::= 0 | 1 | ... | 255
<position>    ::= pt(x:<column> y:<row>)
<row>         ::= 1 | 2 | ... | Input.nRow
<column>      ::= 1 | 2 | ... | Input.nColumn
<life>        ::= 0 | 1 | ...                      %% any positive integer
<score>       ::= ... | -1 | 0 | 1 | ...           %% any integer
<state>       ::= on | off
```

7.2 Configuration file (Input.oz)

The input file allows you to give the parameters of the game. The input is accessible from any of the other modules. It gives:

- Meta information about the game: boolean IsTurnByTurn gives the type of game (true if turn by turn, false if simultaneous), boolean UseExtention indicate if the game is played with (true) or without (fals) extension (an extension is a feature in the game rule not defined in the mandatory part, i.e. shield bonus,...) and boolean PrintOK indicate if you want the debug print only when there is an error (false) or in any case
- Map information: NbRow and NbColumn defines the size of the map, Map is a list of list describing the map (list of the rows, each of them represented as a list), each square being described by a number (0 = simple floor, 1 = wall, 2 = box with point, 3 = box with bonus, 4 = floor with spawn).

- Description of the players: `NbBombers` describes the number of players, `Bombers` is a list giving the name of the bombers used and `ColorBombers` is the list of their colors.
- Parameters of the game: `NbLives` gives the initial number of lives of each of the players, `NbBombs` gives the initial number of bombs of each of the players, `ThinkMin` (resp. `ThinkMax`) gives, in millisecond, the minimum (resp. maximum) time of thinking of the players (used only in simultaneous), `Fire` gives the propagation distance of the fire, `TimingBomb` gives the number of turn before the explosion of a bomb (used only in turn by turn) and `TimingBombMin` (resp. `TimingBombMax`) gives, in millisecond, the minimum (resp. maximum) time for a bomb to explode (used only in simultaneous)

7.3 Controller file (`Main.oz`)

This file contains the main core of the project. This is the file that is responsible to launch the game and coordinate between all the other.

What should be done in this file:

1. Create the port for the GUI and initialise it.
2. Create the ports for the players using the `PlayerManager` and assign its unique ID.
3. Set up the players
4. Launch and coordinate the game (turn by turn/simultaneous)

7.4 Graphical User Interface (`GUI.oz`)

The accepted messages are:

- `buildWindow` : Create and launch the window (no player on it).
- `initPlayer(ID)` : Initialize the `<bomber>` ID (doesn't place the bomberman but just inform the GUI of it's existence, allow to create the score place initialised to 0 and the lives counter initialised to `Input.nbLives`). For a given ID, this should only be used once.
- `spawnPlayer(ID Pos)` : Spawn the `<bomber>` ID at `<position>` Pos. The bomberman should be displayed on the board when sending this message.
- `movePlayer(ID Pos)` : Move the `<bomber>` ID at new position `<position>` Pos.
- `hidePlayer(ID)` : Hide the `<bomber>` ID. This removes the bomberman from the screen.
- `spawnBonus(Pos)` : Spawn the bonus at `<position>` Pos. The bonus should be displayed on the board when sending this message.
- `hideBonus(Pos)` : Hide the bonus at `<position>` Pos. This removes the bonus from the screen.
- `spawnPoint(Pos)`, `hidePoint(Pos)` : Same as for bonuses, but for points.
- `spawnFire(Pos)`, `hideFire(Pos)` : Same as for bonuses, but for fire.
- `hideBox(Pos)` : Same as for bonuses, but for boxes. Boxes are already on the board at initialisation (reading of the map from the input to know the kind of box and the positions).
- `lifeUpdate(ID Life)` : Change the value of the counter for the number of lives left for `<bomber>` ID to the new number of `<life>` Life (put the new value Life as number of lives left for the bomberman).
- `scoreUpdate(ID Score)` : Change the value of the counter for the score for `<bomber>` ID to the new number of `<score>` Score (put the new value Score as score for the bomberman).
- `displayWinner(ID)`: Inform the end of the game, giving the `<bomber>` ID of the highest score bomberman.

Unknown messages should be simply discarded.

Remark: The grid for displaying the board works as a matematical matrix. Top left square is $(1,1)$, going from left to right increase the column number, going from top to bottom increase the row number, square (X,Y) is at the intersection of the X^{th} column and Y^{th} row of the grid.

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

This file is responsible for facilitating the selection of the player (creation of the right port for the player) following the type of the players given in the input file. This file should be the only one you need to modify in order to add another group player available (add the name of the functor in the import part and complete the case part to recognise the name).

7.6 Bomber (`PlayerXXXname.oz`)

The bomber is an agent with two main states: `on` the board or `off` the board (initial position). He get on the board when receiving the `spawn(ID P)` message, can only react to the `action(ID P)` message if he is on the board and pass out of the board when receiving the `gotKilled(ID NewLife)` message.

The accepted messages are:

- `getId(?ID)`[2]: Ask the bomber for its `<bomber>` ID.
- `getState(?ID ?State)`: Assign the `<bomber>` ID and its `<state>` State.
- `assignSpawn(Pos)`: Assign the `<position>` Pos as the spawn of the bomber. This action should only be called once per game.
- `spawn(?ID ?Pos)`: Spawn the bomber on the board. The bomber should answer its `<bomber>` ID and its `<position>` Pos (which should be the same as the one assigned as spawn). This action is only done if the bomber is off the board and has still lives. It places the bomber on the board. ID and P should be bound to null if the bomber is not able to spawn (no more lives back or already on the board).
- `doaction(?ID ?Action)`: Ask the bomber to do its next action. Two actions are possible: Either move to an adjacent floor tile (Action is bounded to `move(Pos)` where Pos is of type `<position>`) , either drop a bomb (Action is bounded to `bomb(Pos)` where Pos is of type `<position>`). Dropping a bomb is only possible if the player has at least one bomb in stock. It decrease the number of bombs in stock by one. Main controller should state the player (using an `add(bomb 1 Res)`) message that it can use the bomb again after it has exploded. An action is only done if the bomber is considered on the board, if not, ID and Action should be bound to null.
- `add(Type Option ?Result)`: Notify the bomber of recieving an additionnal "item". Result is used to return the new value of the counter. Some usage (Option may be used :
  - `add(bomb 1)` (Option being always 1 in the mandatory version): bomber gains one bomb due to the explosion of its previously placed bomb (or due to a bomb bonus if such bonus is implemented)
  - `add(point 1)` (Option being a positive integer in the mendatory version with values given by the Input file): bomber gains one point due to walking on a point or due to a point bonus
  - `add(life 1)`: bomber gains one additional life due to a life bonus (if such bonus is implemented)
  - `add(shield 1)`: bomber gains on additional invulnerability shield (if such bonus is implemented). The shield allows to not die at the next `gotHit` message.

  Types `bomb` and `point` are mendatory to be supported. Others are inspirations.
- `gotHit(?ID ?Result)`: bomber has been hit by fire. The action put him off the board (except if the shield bonus is implemented and he possess one). Bound ID to the `<bomber>` ID and Result to `death(NewLife)`, with NewLife being the new `<life>` value, if the player goes off the board. Bound ID and Result to null if it doesn't go off the board or if it was already off the board when recieving the message.
- `info(Message)`: Inform the player about some events. The mendatory message are:

[2]don't write the ? in your code, it's just to inform you here in the specification, that the sender should put an unbound variable to get a result

- `spawnPlayer(ID Pos)`: Player `<bomber>` ID has spawn in `<position>` Pos
- `movePlayer(ID Pos)`: Player `<bomber>` ID has move to `<position>` Pos
- `deadPlayer(ID)`: Player `<bomber>` ID has died
- `bombPlanted(Pos)`: Bomb has been planted at `<position>` Pos
- `bombExploded(Pos)`: Bomb has exploded at `<position>` Pos
- `boxRemoved(Pos)`: Tile at `<position>` Pos who previously had a box on it is now a floor tile

The bombers have access to the Input file. Therefore they know the map (and the location of other spawns, boxes, walls, bonus boxes, etc.)

7.7 Utils (`Projet2019util.ozf`)

This functor contains two filters: one to check the input messages received by the GUI, the other to check the input messages received by a player. An example of their use is given in the given files.

7.8 Basic player (`Player000bomber.ozf`)

This functor contains a basic player who will move randomly on the board and will drop a bomb (probability of 0.1 to drop a bomb, 0.9 to move). It may help you to develop your main controller.