Ozploding bozmbs – Bomberman in Oz

Alexandre Gobeaux and Gilles Peiffer

***Abstract*—In this paper, the methodology and design choices behind our implementation of the Bomberman game in Oz are given. Both the turn-by-turn part of the game controller as its simultaneous part are explained, and an attempt is made to clarify why the authors took certain decisions with regards to how the final product works. On top of that, an algorithm was implemented in order to control the players' moves, reacting to various messages it receives containing either requests or information about the game. This was done in at various levels of complexity, building increasingly efficient players. On top of the mandatory parts of the project, various supplementary features and embellishments are also included in the implementation.**

In order to validate our product, interoperability tests were also carried out with both the reference player and other teams' players. This paper presents a brief summary of the results of these tests.

I. Introduction

BOMBERMAN is a famous game.

II. Controller structure

THE controller in `Main.ozf` can play the game in two modes: turn-by-turn of simultaneously. We mainly detail the workflow of the turn-by-turn controller, as the simultaneous controller is very similar to this for the main part.

We use the word "broadcast" to signify "send to all players".

A. Turn-by-turn controller

When the game starts, the controller starts by creating the window, and creates a list of the players and their ports thanks to `Input.ozf` and `PlayerManager.ozf`. We then create the map and memorize each player's spawn position, and alert each player of theirs using the `assignSpawn(Pos)` message. Once the GUI is done creating the map, we call the main function of the controller, which starts running the game in turn-by-turn mode, recursively for each round.

This function starts by determining whether the game is over by checking if any of the endgame conditions are met. The controller then checks if there is any fire that needs to be hidden and explodes bombs that run out of time, broadcasting a `bombExploded(Pos)` message if needed. If any players get hit by the blast, a `gotHit(ID Result)` message is sent to them, and information about the death is broadcast to all players. The controller also remembers who died and checks which boxes got destroyed. Information about this is broadcast with the appropriate message, and the endgame conditions are checked again. Finally, the controller updates the map. This raises the question of how to indicate bonuses and points laid bare by explosions; in order to handle this case, the controller defines two new tiles: a point tile with value 5 and a bonus tile with value 6. When a player moves to one of these, their value is changed back to 0, to simulate picking up the reward.

In order to avoid players missing their turn because they got hit by fire before getting a chance to play, we have to respawn the players that died yet still have at least one life left. We broadcast the spawn information to all players and update the list of positions. To avoid sending `doaction(ID Action)` messages to dead players, the controller maintains a counter (implemented with a stream). It then checks whether the player's state is `on`, skipping the player if not, sending a `doaction(ID Action)` message if it is. The player then determines what its next action will be. If the action matches `move(Pos)`, `Main.ozf` checks if any bonuses or points are picked up, and updates the list of positions. If the action matches `bomb(Pos)`, then the controller tells the GUI to spawn a bomb at the specified position. It then sends its blast time on the bomb port. The action is broadcast to all players regardless of the action. After all this, the function is called recursively and the next turn begins.

Alexandre Gobeaux – 42191600
E-mail: alexandre.gobeaux@student.uclouvain.be
Gilles Peiffer – 24321600
E-mail: gilles.peiffer@student.uclouvain.be

B. Simultaneous controller

For the simultaneous controller, we had to implement random delays in the players, to account for "thinking time". This controller is very similar to the turn-by-turn one. Conceptually, it works by creating one global thread, common to all players, and then creating a separate thread for every player, sending a `doaction(ID Action)` message to each, and waiting for a reply using the `Wait` function. Once this reply is received, it is then sent to a stream on the main thread and handled in a similar way to how it was handled by the turn-by-turn controller.

III. Player structure

A. General structure

In order to play the game intelligently, players have to store all relevant information about a game; in order to do this in a clean manner, the authors opted to use record structures, which can easily be modified with the `AdjoinList` function. These records change vary for more advanced players, as they need to store more data about the game in order to make use of their intelligent strategies, but the records always contain at least the following fields:

- `id`: ID of the player;
- `state`: state of the player;
- `lives`: number of remaining lives;
- `pos`: current position;
- `sppos`: assigned spawn position;
- `bombs`: number of bombs owned;
- `map`: map of the game;
- `score`: number of points.

Players have to respond to different messages they can get from the controller, in the form of a stream of instructions. In order to handle each of these messages, multiple so-called "handler functions" were created, one for each possible instruction. Most of the messages that a player can receive in its stream are relatively straightforward; in this paper, only the most important instruction, `doaction(ID Action)` is explained.

The `doaction(ID Action)` message asks the player for its ID and its next action (place a bomb at its current location or move to a neighbouring location). Multiple players were implemented, with the difference between them lying mainly in the way the "optimal" action is computed. These differences are explained in Section III-B.

B. Decision algorithms

1) Basic player: In our basic random player, `Player001Kardashian.ozf`, the decision algorithm looks like this:

- If the player has no bombs left, move to a random neighbour with uniform probability over the acceptable moves.
- If the player has bombs left, the player has a $10\,\%$ chance to drop a bomb, and a $90\,\%$ chance to move to a randomly chosen acceptable neighbour.

This strategy is not very efficient: the player does not avoid bombs, and does not hunt for points.

2) Advanced player: Our next player, `Player001Tao.ozf`, is slightly more advanced; its decision algorithm tries to avoid standing in dangerous areas, i.e. too close to bombs. This means that the advanced player has to keep track of where bombs are located. In order to do this, one must simply add an argument to the `summary` record: `bomblist`, which stores a list of bombs that are currently on the map. When the player has to determine its next action, it uses the following algorithm:

- If the player has no bombs left, move to the least dangerous acceptable neighbour, where the danger rating of a tile is found by initialising it to zero, and then adding points if the tile is in the blast radius of a bomb (taking into account that walls and boxes stop fire from spreading, as well as the distance from the bomb).
- If the player has bombs left, the player has a $10\,\%$ chance to drop a bomb, and a $90\,\%$ chance to move according to the algorithm above.

This strategy is more successful than the basic player, and most of the time manages to avoid bombs. It does not however hunt for points or bonuses, and can make wrong choices when trying to avoid bombs (forcing itself into a dead end, for example).

3) Intelligent player: The intelligent player, `Player001Turing.ozf`, builds further on what the advanced player does, by adding the ability to intelligently search for points and bonuses, with the latter being preferred in most cases. In order to track where points and bonuses are located, one has to take care of treating the `boxRemoved(Pos)` messages. The algorithm is the following:

- If the player has no bombs left, move to the least dangerous acceptable neighbour, where the danger rating of a tile is found by initialising it to zero, and then adding points if the tile is in the blast radius of a bomb (taking into account that walls and boxes stop fire from spreading, as well as the distance from the bomb). If multiple neighbours are equally safe, run an iterative deepening depth-first search for points and one for bonuses. If possible, the player tries to go for a bonus first, given that the distance to the next bonus is not too big compared to the distance to the next point[1]. If there is a point target and no bonuses, the player moves towards it on the shortest path. Finally, if there are no special targets, the player chooses a random neighbour from the safe options.
- If the player has bombs left, it tries to place a bomb given that it is next to a box (in order to lay bare its content). Otherwise, it moves towards a position according to the algorithm above.

This strategy works very well, and consistently came out as one of the top AIs when competing with other teams.

IV. Interoperability

In order to validate our implementation, we carried out interoperability tests with various groups. These tests helped us to identify misunderstandings in the project specification, as well as find some tricky bugs.

A. Group 55 – Mulders, Semerikova

1) `Player055Random`*:* While trying for interoperability with this group, we were reminded that our player is not allowed to place two bombs on the same tile. In order to fix this mistake, we had to rework the way the list of bombs is kept. Thinking about this also helped us to solve another issue we had when playing simultaneously; with the new design, our implementation was no longer specifically tailored to a certain game mode, i.e. turn-by-turn. Another error we found is that the advanced players did not always bind an ID when returning an Action. Since our `Main.ozf` was very robust and did not actually need these values as they were being kept track of, the error never showed on our own computers.

Another issue we ran into while testing our players with this group was that after some rounds, our player would stop planting bombs, despite being under the right circumstances to do so. When investigating the code, this came down to a mismatch in specifications between the initialisation of the list of bombs and the `case` statement trying to access it.

2) `Player005Clever`*:* As we were playing a game against their advanced player, we suggested a different, more efficient approach. There were no bugs with their player or their controller, as far as we could see.

V. Extensions and optional parts

Some of the proposed extensions and improvements were implemented, and the authors added some of their own:

- As explained in Section III-B, one of our players, `Player001Turing.ozf`, uses a fairly advanced algorithm to determine its next action.
- The GUI has been embellished to resemble the user interface of the Minecraft game.
- The synchronization between the GUI and the controller has been automated, to avoid the game starting while the GUI is still loading.

VI. Conclusion

Working on this project has helped us to better understand

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[1]Heuristically, we found a ratio of 3 to work fairly well.