LINGI1131 - Project 2018
PacmOz

Hélène Verhaeghe & Guillaume Maudoux

Version 6: 18-04-18

1 Introduction

Everybody knows Pacman, a game from 1980, where the pacman (yellow ball) has to collect points and bonus while avoiding the ghost chasing him.

Your mission will be to implement your own message passing agents for the pacman and the ghosts.

2 Rules of Pacman

Our rules will differ a bit from the original ones. First, we will have two modes: turn by turn and simultaneous. Some actions are specific to one or the other. Secondly, the game may be multi-player (multiples pacmans and multiples ghosts).

The game starts with the pacman spawning at a random available spawn for pacman and the ghosts spawning at a random available spawn for ghost. The spawn will be assigned to the player (he should respawn at the same spawn if killed).

Points are produced on simple walkable square (labeled with 0) while bonus are produce on a more specific walkable square (labeled with 4). If a pacman walks on the square and the points/bonus are available, then it gets it. The square doesn't instantaneously create the new points/bonus, it waits a given time (given in `Input.oz`) before having a new point/bonus available. The bonus swaps the role for a given amount of time (given in the `Input.oz`) the pacmans become the hunters and allow him to kill ghost. If a second bonus is caught while already in hunt mode, it only reset the timing. At the beginning of the game, all the bonus and points are available on the board (don't need to wait before spawning them the first time).

Pacmans and ghosts can move in four directions (north, south, east, west). When on a given square, they can move in a given direction only if the adjacent square in that direction is not a wall. The map is self wrapping. This means that if a pacman is on a given square of the last column, he can still go east, arriving on the first column (and still on the same row). If there is a wall on the first column, then it's not possible to go east.

The game end when all pacmans are dead. The winner is the one with the highest score.

2.1 Some special cases

- multiple pacmans can be at the same time on the same square
- multiple ghosts can be at the same time on the same square
- pacman(s) and ghost(s) can't be at the same time on the same square (in simultaneous mode, they can be some borderline case like a ghost moving from (0,0) to (0,1) while a pacman moves from (0,1) to (0,0), it is possible that they don't meet. If you want, you can create an agent responsible to sequensing the movement to avoid such situations).
- each bonus/point can be get by one player at the time (if two players simultaneously enter the same square, the one getting the bonus/point is chosen at random)

- if a pacman is on the same square as multiple ghost, it gets killed only once, the killer ghost is chosen randomly (same if during hunt mode and a ghost meet multiple pacmans)
- if a ghost arives on a square with multiple pacmans (in classical mode), he kills all the pacmans (same in hunting mode if one pacman meets multiples ghosts)

3 Game general behaviour

3.1 Turn by turn

In turn by turn, the first part is to determine the order (at random) between all the ghosts and the pacmans (ex: pacmans and ghost are mixed), 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 a move if it is possible (if player is on the board, if not, just skip its turn).

The game ends when all pacman player have lost all their lives.

3.2 Simultaneous

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

Then each player is launch 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 done before a significant `move` message (message `move` received when on the board) is processed by the player agent.

The game ends when all pacman player have lost all their lives.

4 Specifications

The following subsections will give you the specification for all the agent you have to implement. We ask you to follow theses in order to make your agents interoperable between each other.

4.1 Types used (EBNF formula)

```
<pacman>      ::= pacman(id:<idNumP> color:<color> name:<name>)
<idNumP>      ::= 1 | 2 | ... | Input.nbPacman
<ghost>       ::= ghost(id:<idNumG> color:<color> name:<name>)
<idNumG>      ::= 1 | 2 | ... | Input.nbGhost
<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

<lives>       ::= 0 | 1 | ... | Input.nbLives
<mode>        ::= classic | hunt
```

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

- `isTurnByTurn`: true if the game is set up turn by turn, false if simultaneous
- description of the map:
  - `nRow`: number of row of the map
  - `nColumn`: number of column of the map

  - `map`: description of the map as a list of list of integer (0 = walkable place and spawns for point, 1 = wall (not walkable), 2 = walkable and spawn(s) for pacman(s), 3 = walkable and spawn(s) for ghost(s), 4 = walkable and spawn(s) for bonus). If the number of spawn is N and the number of player to put on the spawns is M, the maximum number of player per spawn is $\lceil \frac{M}{N} \rceil$ (if you have 3 ghosts and 4 ghost's spawns, you have maximum $\lceil \frac{3}{4} \rceil = 1$ ghost per spawn, 6 pacman and 2 spawns, maximum $\lceil \frac{6}{2} \rceil = 3$ pacman per spawn).

An example of map:

```
NRow = 5
NColumn = 10
Map = [[0 0 0 0 0 3 0 0 0 0]
       [0 1 1 1 0 1 1 0 1 0]
       [0 0 0 1 0 0 1 0 1 0]
       [0 1 0 1 1 0 1 0 1 0]
       [4 1 0 2 0 0 0 2 0 0]]
```

The number of spawns for pacman and ghosts is at least of one, but there can be multiple. Each non-wall square can be walk over.

- respawn time of the different items and players:
  - `respawnTimePoint`: in turn by turn, number of turn to wait until new spawn of points, in simultaneous, number of seconds to wait to have a new spawn of points
  - `respawnTimeBonus`: in turn by turn, number of turn to wait until new spawn of bonus, in simultaneous, number of seconds to wait to have a new spawn of a bonus
  - `respawnTimePacman`: in turn by turn, number of turn to wait until new spawn of pacman, in simultaneous, number of seconds to wait to have a new spawn of pacman
  - `respawnTimeGhost`: in turn by turn, number of turn to wait until new spawn of ghost, in simultaneous, number of seconds to wait to have a new spawn of ghost
- rewards and penalties:
  - `rewardPoint`: number of point the pacman gets when he catch a basic reward
  - `rewardKill`: number of point the pacman gets when he kills a ghost
  - `penalityKill`: number of point the pacman looses when he gets kill by a ghost
- `nbLives`: number of lives of each pacman
- `huntTime`: in turn by turn, number of (complete) turns to wait until the mode is set to classical again , in simultaneous, number of seconds being the duration of the hunt mode
- `thinkMin` and `thinkMax`: in simultaneous, bounds of the number of millisecond the player (pacman or ghost) has to wait before answering a significant `move` message

To avoid specific complex situations, whe define the specific constraint`respawnTimePacman`$*1000 \geq$ `thinkMax` $+2000$ and `respawnTimeGhost`$*1000 \geq$ `thinkMax` $+2000$.

4.3 Graphical User Interface (`GUI.oz`)

The accepted messages are:

- `buildWindow` : Create and launch the window (no player on it).
- `initPacman(ID)` : Initialize the `<pacman>` ID (doesn't place the pacman 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.

- `spawnPacman(ID P)` : Spawn the `<pacman>` ID at `<position>` P. The pacman should be displayed on the board when sending this message.
- `movePacman(ID P)` : Move the `<pacman>` ID at new position `<position>` P.
- `hidePacman(ID)` : Hide the `<pacman>` ID. This removes the pacman from the screen.
- `initGhost(ID)` : Initialize the `<ghost>` ID (doesn't place the ghost but just inform the GUI of it's existence). For a given ID, this should only be used once.
- `spawnGhost(ID P)` : Spawn the `<ghost>` ID at `<position>` P. The ghost should be displayed on the board when sending this message.
- `moveGhost(ID P)` : Move the `<ghost>` ID at new position `<position>` P.
- `hideGhost(ID)` : Hide the `<ghost>` ID. This removes the ghost from the screen.
- `initBonus(P)` : Initialize the bonus at `<position>` P (doesn't place the bonus but just inform the GUI of it's existence). For a given `<position>` P, this should only be used once.
- `spawnBonus(P)` : Spawn the bonus at `<position>` P. The bonus should be displayed on the board when sending this message.
- `hideBonus(P)` : Hide the bonus at `<position>` P. This removes the bonus from the screen.
- `initPoint(P)` : Initialize the point at `<position>` P (doesn't place the point but just inform the GUI of it's existence). For a given `<position>` P, this should only be used once.
- `spawnPoint(P)` : Spawn the point at `<position>` P. The point should be displayed on the board when sending this message.
- `hidePoint(P)` : Hide the point at `<position>` P. This removes the point from the screen.
- `lifeUpdate(ID L)` : Change the value of the counter for the number of lives left for `<pacman>` ID to the new number of `<lives>` L (put the new value L as number of lives left for the pacman).
- `scoreUpdate(ID S)` : Change the value of the counter for the score for `<pacman>` ID to the new number of `<score>` S (put the new value S as score for the pacman).
- `setMode(M)`: Inform the new `<mode>` M (indicate the change of mode, in `classic` mode, ghosts have their color given in the `<ghost>` type, when passing to `hunt` mode, they change to the same color.
- `displayWinner(ID)`: Inform the end of the game, giving the `<pacman>` ID of the highest score pacman.

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.

4.4 Player pacman (`PacmanXXXname.oz`)

The pacman is an agent with two main states: on the board or not. Initially he is not on the board. He get on the board when receiving the `spawn(ID P)` message, can only react to the `move(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)`[1]: Ask the pacman for its `<pacman>` ID.
- `assignSpawn(P)`: Assign the `<position>` P as the spawn of the pacman.
- `spawn(?ID ?P)`: Spawn the pacman on the board. The pacman should answer its `<pacman>` ID and its `<position>` P (which should be the same as the one assigned as spawn. This action is only done if the pacman is not on the board and has still lives. It places the pacman on the board. ID and P should be bound to null if the pacman is not able to spawn (no more lives back).
- `move(?ID ?P)`: Ask the pacman to chose its next `<position>` P (pacman is thus aware of its new position). It should also give its `<pacman>` ID back in the message. This action is only done if the pacman is considered alive, if not, ID and P should be bound to null.
- `bonusSpawn(P)`: Inform that a bonus has spawn at `<position>` P
- `pointSpawn(P)`: Inform that a point has spawn at `<position>` P
- `bonusRemoved(P)`: Inform that a bonus has disappear from `<position>` P, doesn't say who eat it.
- `pointRemoved(P)`: Inform that a point has disappear from `<position>` P, doesn't say who eat it.
- `addPoint(Add ?ID ?NewScore)`: Inform that the pacman has gain the number of points given in Add, and ask you your `<pacman>` ID and the NewScore you have.
- `gotKilled(?ID ?NewLife ?NewScore)`: Inform that the pacman has lost a life and pass it out of the board. Ask him its `<pacman>` ID, its new number of lives in NewLife and its new score (as you lose point when been killed). This action makes the pacman in a dead state.
- `ghostPos(ID P)`: Inform that the ghost with `<ghost>` ID is now at `<position>` P.
- `killGhost(IDg ?IDp ?NewScore)`: Inform that the ghost with `<ghost>` IDg has been killed by you. Ask you your `<pacman>` IDp back and your NewScore (since killing a ghost make you gain points).
- `deathGhost(ID)`: Inform that the ghost with `<ghost>` ID has been killed (by someone, you or another pacman).
- `setMode(M)`: Inform the new `<mode>` M.

The pacman is initially not on the board and does not have to wait the require respawn time at the first spawn.

In simultaneous, it is stongly advised to find a way (easy normaly) so your pacman continues to handle messages while thinking (in case of death during thinking, he should be capable to accept his death and not continue to move).

4.5 Player ghost (`GhostXXXname.oz`)

As the pacman, the ghost is an agent with two main states: on the board or not. Initially he is not on the board. He get on the board when receiving the `spawn(ID P)` message, can only react to the `move(ID P)` message if he is on the board and pass out of the board when receiving the `gotKilled()` message.

The accepted messages are:

- `getId(?ID)`: Ask the ghost for its `<ghost>` ID.

[1] 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

- `assignSpawn(P)`: Assign the `<position>` P as the spawn of the ghost.
- `spawn(?ID ?P)`: Spawn the ghost on the board. The ghost should answer its `<ghost>` ID and its `<position>` P (which should be the same as the one assigned as spawn. This action is only done if the ghost is not on the board. It places the ghost on the board.
- `move(?ID ?P)`: Ask the ghost to chose its next `<position>` P (ghost is thus aware of its new position). It should also give its `<ghost>` ID back in the message. This action is only done if the pacman is considered on the board, if not, ID and P should be bound to null.
- `gotKilled()`: Inform the ghost that it had been killed and pass it out of the board.
- `pacmanPos(ID P)`: Inform that the pacman with `<pacman>` ID is now at `<position>` P.
- `killPacman(ID)`: Inform that the pacman with `<pacman>` ID has been killed by you.
- `deathPacman(ID)`: Inform that the pacman with `<pacman>` ID has been killed (by someone, you or another ghost).
- `setMode(M)`: Inform the new `<mode>` M.

The ghost is initially not on the board and does not have to wait the require respawn time at the first spawn.

In simultaneous, it is stongly advised to find a way (easy normaly) so your ghost continues to handle messages while thinking (in case of death during thinking, he should be capable to accept his death and not continue to move).

4.6 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 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).

4.7 Main controller (`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)

5 Interoperability

If you follow the specification in a right way, you will be able to test your implementation with other pacmans and ghosts from other groups. We ask you to test with at least 5 other different players (the 2 basic players given doesn't count in the 5) from at least 2 different groups. Feel free to test with more.

When sharing your player (ghost/pacman) 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 the player you have tested should be given in the report with some remarks on how it went, if it helped you find mistakes,...

6 Submission and logistic details

6.1 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.

6.2 Summary on what to do

The project has to be done by groups of two people (groups of one are tolerated but not recommended). No plagiarism will be tolerated (you can help another group but never give share your code).

6.2.1 Mandatory part

Your project has to implement the following mandatory points:

- Create your own `GUI.oz` file for the GUI or modify the one given
- Create a `Main.oz` file for the game controller that should be able to handle both turn by turn and simultaneous game modes (it should adapt according to the variable specified in the `Input.oz` file).
- Create at least one Pacman player and at least one Ghost player. At most one of them should be a bit more than just random behaviour.
- Update the `PlayerManager.oz` for your own players.
- Test the interoperability of your project with at least 5 players from at least 2 different groups, at least one ghost and one pacman over the 5 players. Notify in your report with groups and which player exactly you have tested. Sharing the players is only allowed by sharing the .ozf compiled file.
- Write a `Report.pdf` of maximum 5 pages of content (cover page and table of content doesn't count) explaining your implementation and detailing what you have done (the behaviour of your two player, your main controller, ...).

6.2.2 Optional part

Your project may implement other improvements following your creativity. They should be mentioned and explained in the report. Some ideas:

- Create other Pacman players.
- Create other Ghost players. You can inspire yourself from the original ghosts in the genuine Pacman game (internet may help you for that).
- Add sounds to the game.
- Create map generator.

- Create a human player to manage to play with the keyboard as input.
- ...

Remark: Optional parts are here to help you having more points, but they won't bring points to a project where some mandatory parts are missing.

6.3 Submission

Your submission will be done on INGInious. You have to submit a .zip file containing the different files of your project.

- your `Main.oz` file containing the code of your controller.
- at least one `PacmanXXXname.oz` file (where XXX is your group number (ex: group 42 has to put 042), and name is the name of this pacman (ex: basic, Hector, ...) allowing to differentiate your different pacmans.
- at least one `GhostXXXname.oz` file (where XXX is your group number (ex: group 42 has to put 042), and name is the name of this ghost (ex: basic, Hector, ...) allowing to differentiate your different ghosts.
- your `Input.oz` file with an original map (meaning not the example one and not the same as another group).
- the ghost and pacman `.ozf` files of the other group you have tested with.
- your `PlayerManager.oz` file ready with your player and the ones of the groups you have tested with.
- a make file allowing us to compile your project.
- your `Report.pdf` file.

The deadline is due for Friday the 27th of April 2018 6pm (end of Week 10). You will be asked to do a small demo during the Week 11. Schedule and informations about the demo will arrive later.