LINFO1104: LethOz Company

Academic year 2023 - 2024

Teacher : Peter Van Roy
Courses : LINFO1104
Collaborators :
Crochet Christophe,
Clément Delzotti,
Evrard Colin

Contents

1 Overview

LethOz Company is a strategic multiplayer game in which players control spaceships players control spaceships in an asteroid field, collecting resources on behalf of the large intergalactic mining company *Large Ethereum Organization of Zadristan*, better known by its nickname *LethOz*. The aim is to develop the company in order to collect waste, avoid danger and strategically use bonuses to outperform opponents. outwit opponents. This project uses the multi-paradigm nature of the Oz programming language programming language, using only declarative programming, game state management and the application of dynamic effects.

This project use the best multi-paradigm nature of the Oz programming language, only using declarative programming, game state management, and dynamic effect application.

1.1 Multiplayer Support

The game supports multiple players, each controlling their ship with distinct strategies and objectives.

2 Game Context

- The game is set in a dynamic and hostile space environment, with each cell in the grid representing space filled with scraps, asteroids, or special bonuses.
- Players control ships to gather scraps for The Company, avoid or strategically place seismic-charges to disrupt opponents, and collect bonuses to enhance their capabilities.
- As a player, one of your objectives is to collect as many scraps as possible. This comes at a cost however, as every scrap will take space. These will add up at the end of your space ship's cargo, in a way that is very similar to modern days cargo trains and wagons. Therefore, your space ship will need to avoid colliding with it's own cargo and of course other ships !

3 Core Functionalities

1. **Ship Movement and Strategy Decoding**: Implement logic for navigating ships through the hostile space environment, including a strategy decoding mechanism allowing players to predefine their navigational tactics.
2. **Dynamic Effects Application**: Design a variety of effects such as speed boosts, shields, and warp capabilities that can be applied to ships to affect gameplay.
3. **Game State Management**: Develop the backend logic to manage game progression, including tracking mined resources, managing seismic-charge placements, and resolving player interactions.

4 Instructions

The project must be completed in pairs, with each team required to implement the strategy decoding and the three essential additional effects outlined below.

Once you've mastered the fundamental aspects, you're encouraged to enhance the game by incorporating a **minimum of two additional features**.

While there's no limit to the number of extensions you may introduce, *priority should be given to fulfilling the core project requirements first.*

Remember, adding numerous extensions cannot compensate for a project that's fundamentally flawed or poorly executed.

Your evaluation will be based on the quality of both your code and your report. We expect your code to be correct, well-organized (broken down into smaller, well-commented sub-problems), and as efficient as possible (consider using tail recursion). You're advised to stick to the declarative features of the language, **no Cell or Array**. You can use the standard library (but they need to be declarative).

If you deviate from these guidelines, please explain and justify your decisions in the accompanying report along with your code and test scenario.

Code sharing between different teams is strictly prohibited, except for scenarios. The test scenario you submit must be unique and not replicate any from other teams. However, discussing and exchanging ideas is permitted and encouraged.

4.1 Base objectives

The game unfolds on a grid measuring 24 squares in width by 24 squares in height.

- Each square is identified by coordinates ((x, y)) such that $(1 \leq x \leq 24)$ and $(1 \leq y \leq 24)$
- The coordinate $(1, 1)$ marks the upper left corner, or the north-west, of the play area.

Each grid square may be occupied by parts of a spaceship, asteroids (serving as obstacles), bonuses, or combinations thereof. Below is an example of what the game might look like at any given moment:
Gameplay progresses in several steps, during each of which the following actions occur:

1. Spaceships deploy seismic-charges as programmed.
2. seismic-charge timers on the grid tick down, and those reaching zero detonate.
3. **Spaceships update based on their current effects and the scheduled instruction for that turn** (this is your intervention point).
4. Bonuses are applied to spaceships, and any destroyed spaceships are identified. A spaceship is destroyed if:
    - It collides with another spaceship, itself, an asteroid, or a wall;
    - It is caught in a seismic-charge explosion;
    - It exits the play area.

At this stage, the game engine also determines if a team can be declared the winner, ending the game and proclaiming victory. A player is a winner if their spaceship is the last one remaining on the grid or it has collected 10 scraps.

Each game's *scenario* defines the starting positions of the spaceships on the grid, as well as the locations of bonuses and any seismic-charges placed from the start. Each *spaceship* occupies several positions at a time, each with an orientation from one of the cardinal directions: `north`, `south`, `west`, or `east`. The first position listed is the spaceship's front, and the last is the rear.

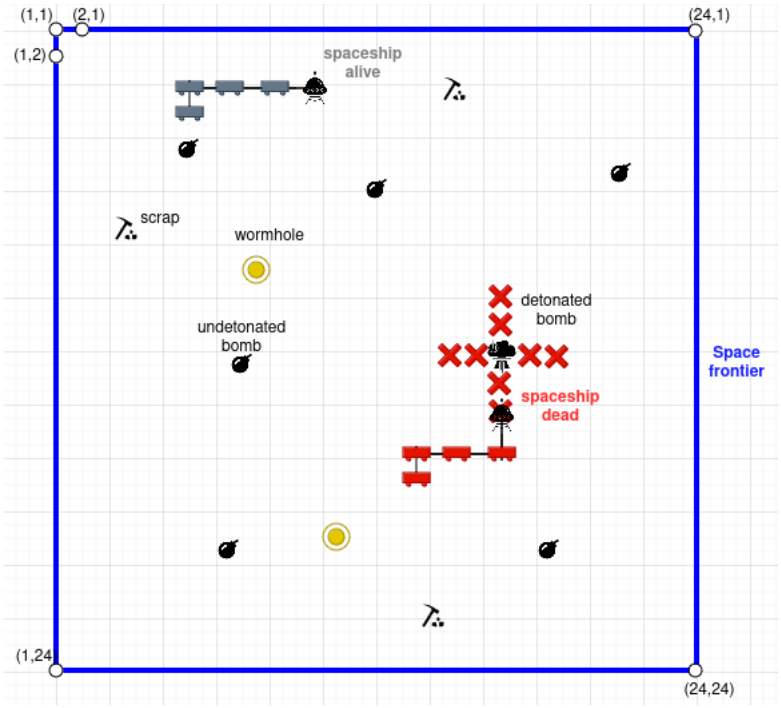

Figure 1: Exemple at a given time

```
P         ::= <integer such as 1 <= x <= 24>
direction ::= north | south | west | east

spaceship ::= spaceship(
  positions: [
    pos(x:<P> y:<P> to:<direction>) % Head
    ...
    pos(x:<P> y:<P> to:<direction>) % Tail
  ]
  effects: nil
)
```

Each *spaceship* follows a predefined strategy at each step, requiring it to execute an instruction: `forward`, `turn(left)`, or `turn(right)`. Executing this instruction involves reorienting the spaceship's front in the desired direction and moving it one square forward in that direction. This is the purpose of your `Next` function.

```
relative_direction ::= left | right
instruction        ::= forward | turn(<relative_direction>)
```

The game engine, for each spaceship, expects a decoded strategy as a **simple list of functions**. Each function takes a single argument, a *spaceship*, and returns a new *spaceship* with updated attributes. The (n^{th}) function in each spaceship's decoded strategy is applied at the game's n^{th} step. You must develop the `DecodeStrategy` function to convert strategies of the following form

```
strategy ::= <instruction> '|' <strategy>
           | repeat(<strategy> times:<integer>) '|' <strategy>
           | nil
```

into a list of functions. Each function must utilize your `Next` function to update the spaceship according to the step's instruction. This approach, including the syntax for repeating sub-strategies with `repeat(...)`, requires careful thought.

Summary Begin by implementing two functions in `Code.oz`:

- `fun {Next Spaceship Instruction} ... end`, applying the `Instruction` on the spaceship `Spaceship` and returning the updated spaceship. The returned spaceship should maintain its size (unless specified otherwise by an effect), have its front reoriented, and move according to the instruction.

  For instance, for the arguments `spaceship(positions:[pos(x:4 y:2 to:east) pos(x:3 y:2 to:east) pos(x:2 y:2 to:east)] effects:nil)` and `turn(right)`, the result should be `spaceship(positions:[pos(x:4 y:3 to:south) pos(x:4 y:2 to:south) pos(x:3 y:2 to:east)] effects:nil)`.

  Details on effects will follow in the next section.

- `fun {DecodeStrategy Strategy} ... end`, with `Strategy` defined as earlier. `DecodeStrategy` transforms the strategy into a list of functions of type `fun {$ Spaceship} ... spaceship(...) end`. Each function updates the spaceship for the instruction at that game step.

  For example, `{DecodeStrategy [repeat([turn(right)] times:2) forward]}` yields a list of 3 functions `fun {$ Spaceship} ... spaceship(...) end`, with the first two applying `turn(right)` to the spaceship and the third applying `forward`. These functions must call `Next`.

Company effects To help you with your task, the Company provided different effects scattered around the map. These effects are added to a spaceship's `effect` list by the game engine when its front reaches a bonus. It's your responsibility to apply these rules and remove them once accounted for by your `Next` function.

The game engine only adds to this list, and rules **must be applied in order**. If multiple effects of the same type are present, only the first one should be considered; others should be ignored and removed.

1. `scrap`: The spaceship extends by one square at the end of its movement. Its front moves as usual.

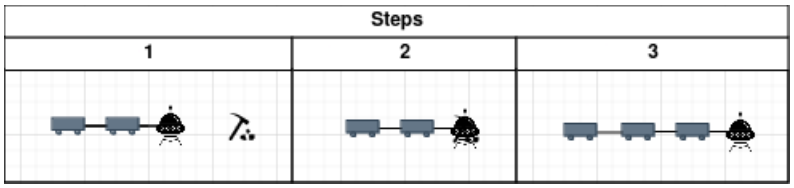

Figure 2: Evolution of a spaceship going forward affected by the `scrap` at step `2`

2. `revert`: The spaceship inverts before executing the move instruction, turning its rear into its front and vice versa, including direction reversal.

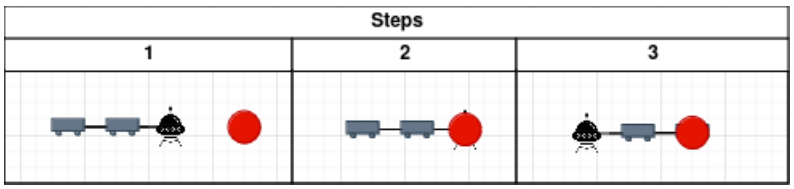

Figure 3: Evolution of a spaceship going forward, affected by the `revert` at step `2`.

3. `wormhole(x:X y:Y)`: The spaceship's front, upon reaching a bonus, is teleported to position `(X, Y)` before executing the move instruction, retaining its orientation.

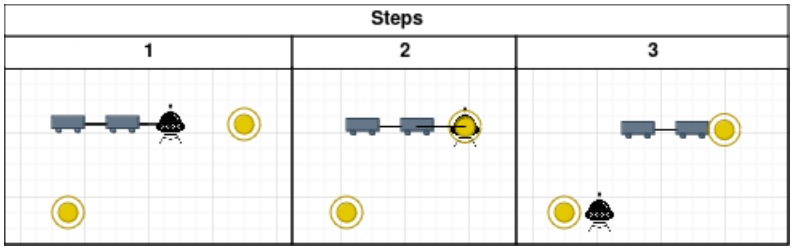

Figure 4: Evolution of a spaceship going forward, affected by the `wormhole(...)` at step 2..

Tips This foundation of the project suggests starting with basic instructions to ensure your *spaceships* correctly respond to the `'scenario_test_moves.oz'` scenario, which lacks any bonuses. Then, tackle the `scrap` rules, followed by `wormhole(...)` and `revert`, testing with their respective scenarios.

However, passing these tests does not guarantee correctness. Experiment with writing your scenarios and testing the game interactively to explore more gameplay possibilities.

Sharing your scenarios on the Moodle forum is encouraged—just not the one you plan to submit.

4.2 Extensions

In accordance with the project guidelines, you are required to develop at least two additional features. While your creativity sets the boundaries, we offer a few suggestions to spark inspiration or provide direction for further innovation.

All the code provided to you is part of the project. This means that you are expected to understand how the project works, which file is responsible for which functionality, and how they interact at the code level. In particular, to test your code and implement new features, you will probably need to read the contents of the file `project_library.oz`, which contains many of the functions useful for the project.

To evaluate your extensions effectively, you will need to either craft a new scenario or modify an existing one, incorporating or changing bonuses as necessary.

These bonuses include:

- attributes such as position, `color` (for visual distinction),
- the rules applied to spaceships,
- and the `target` attribute determining which spaceship is affected upon contact.

Commonly, `catcher` is assigned as the `target` value in many scenarios, applying the bonus to the spaceship that triggers it. Alternatives include `others`, `allies`, `opponents`, and `all`, broadening the potential strategies.

```
color ::= <any colors recognized by Tk, see https://www.tcl.tk/man/tcl8.4/TkCmd/colors.htm>
bonus ::= bonus(position: pos(x:<P> y:<P>)
                color: <color>
                effect: boost | wormhole(x:<P> y:<P>) | reverse | ...
                target: catcher | others | allies | opponents | all
)
```

Wrap-around Grid Consider designing scenarios without peripheral barriers (`walls` attribute set to `false` in your scenario's config (see `scenario_test_moves.oz` for an example of a scenario config), where spaceships seamlessly transition across the grid's edges, reappearing on the opposite side.

This mechanic prevents spaceships from being destroyed upon reaching the boundary, akin to the teleport bonus's effect.

The `shrink` Effect While scrap effects exist, introducing a `shrink` effect could add a new layer of challenge, as spaceships risk becoming too small to maintain viability.

The `shield(N)` Effect Provides temporary invincibility against seismic-charges and collisions.

The `emb(N)` Effect (ElectroMagnetic bomb) This effect immobilizes the spaceship for `N` steps, disregarding any given instructions and potentially overriding other active effects.

The `malware(N)` Effect Inverting `turn(left)` and `turn(right)` instructions for `N` steps could introduce humorous and chaotic elements, especially in multiplayer scenarios.

The `time_warp` Effect A rare bonus that allows a player to rewind time by one turn, undoing the actions of all players. This complex effect would require tracking the state of the game across turns, allowing a rollback to the previous state. It could be limited in use to maintain balance.

Multi-Grid Arenas Expand the gameplay beyond a single grid by introducing multiple interconnected grids or levels that players can move between. Spaceships could find portals that transport them to different grids, each with unique challenges and bonuses.

Broadening the Horizon To fully leverage the game's potential, you're encouraged to draw inspiration from existing scenarios or conceive entirely new effects, strategies, or gameplay elements. Spaceships might have a team identifier (`team`, any Tk-recognized color) and a seismic-charge strategy (`seismicCharge`, a boolean list consumed stepwise to determine bomb deployment).

Your creativity can extend to modifying spaceship attributes via the `Next` function, similar to handling `positions` and `effects`. For instance, a spaceship with an `shield` effect at a turn's end might survive otherwise fatal situations, whereas one afflicted by a `death` effect would be eliminated for the game's duration.

Feel free to invent new instructions or syntactic structures for strategies, ensuring they complement the base game mechanics. These innovations could transform the game into a new experience reminiscent of *TRON* or other iconic games, provided they don't interfere with the core strategies and effects.

5 Submission Guidelines

Submit your project as a zip archive containing the following files:

1. `Code.oz`: Your Oz code implementing the game logic.
2. `Report.pdf`: A detailed report discussing your design choices, encountered difficulties, complexity analysis of your functions, and descriptions of your implemented extensions. The report should not exceed 5 pages.
3. `Scenario.oz`: A fully automated game scenario demonstrating all functionalities and extensions you have implemented.

Ensure your scenario showcases the unique aspects of your project.

5.1 Evaluation Criteria

- **Code Quality**: Your code will be evaluated based on its efficiency and adherence to declarative programming principles.
- **Readability and Clarity of the Code**: Your code will not be automatically evaluated. This means that we will read your code in addition to running it. Just as a law professor expects impeccable grammar and legible writing in an essay, we expect you to consider that your code must be easy to understand **at first glance** by a third party. Illegible and/or incomprehensible code will therefore be penalized. Here are some tips to help you produce clear code.
  1. **Indent and space your code**: Ideally, each control structure adds a level of indentation (similar to Python). Group lines of code that go together, separate groups with one or more line breaks.
  2. **Comment your code**: Whenever the reader of your code might question the purpose of a part of the code, write a concise comment describing what you are trying to accomplish. Similarly, each function/procedure should come with a description of the inputs it expects, and the possible expected outputs. However, try to remain concise, we do not expect you to write the next opus of *Game of Thrones*.
  3. **Break down your code into sub-problems**: Perhaps you are familiar with the concept of *code smell*? It refers to characteristics of the code that indicate your approach is bad. One such indicator is the presence of *gigantic functions* with an absurd number of arguments. A good heuristic to avoid this is to always try to break down your function into sub-problems which themselves will be solved by their own function. The resulting code should not have functions longer than about twenty lines. There can obviously be exceptions like `case of` instructions which can quickly take up many lines.
- **Innovation and Complexity**: Implementations that go beyond the basic requirements and demonstrate innovative thinking will be highly regarded.

5.2 Deadline

Your project must be submitted by the **29th April 2024** at 23h55. Late submissions will be penalized.

6 Additional Notes

- While collaboration on ideas is encouraged, code sharing between groups is strictly prohibited.
- Feel free to use the course forum for discussions and questions. However, avoid sharing specific code implementations.

Good luck, and we look forward to seeing your innovative space mining strategies and game implementations!