Project Milestone 5

CS 4500 Software Development

Due: Thursday, March 11, 9pm

Submission: Create a release on GitHub tagged `p5`. Add a `p5-exec.zip` to the release, containing:

1. The `testState` executable for the testing task
2. Any packages/modules necessary for the above executable to function
3. At least 4 tests in `tests/`

Here is a sketch of the `p5-exec.zip` layout:

```
|-- testState
|-- tests
|   |-- 1-in.json
|   |-- 1-out.json
|   ...
...
```

Submit a ZIP with the following on Handins:

1. For the design task, the file `observer.md` in `Snarl/Planning/`
2. For the programming task, any new or updated *relevant* source files under `Snarl/src`. The Game Manager and Rule Checker implementations should go under a `Game` subfolder.[1]
3. For the testing task, a folder `Snarl/tests/State` with the test harness source code and your test suite for

Here is a sketch of the Handins ZIP layout:

```
Snarl
|-- Planning
|   `-- observer.md
|-- tests
|   `-- State
|        |-- testState.<ext>
|        |-- 1-in.json           <-- these files can be under an additional "tests"
|        ...                         directory, just like in the exec zip
`-- src
    |-- Game
    |   |-- ruleChecker.<ext>
    |   |-- gameManager.<ext>
    |   ...
    ...
```

[1]Ask us on Piazza if you have doubts about how to organize the source tree. Our main concern is some level of uniformity in structure so we can find what we are looking for across many submissions.

Design Task

For the purposes of debugging and presenting a game in progress to the stakeholders, we need a component which will allow viewing a game in progress. Design the interface for such an "observing" component. For this task, design a *local* API, but keep in mind that, eventually, an observing component might connect to the game via a network. I recommend taking a peek at the Observer design pattern for inspiration, or a starting point for the API.

Note that the idea of an observing component is separate from the user interface for the human players, although they might be similar.

Scope: We are looking for data definitions, signatures and purpose statements à la Fundies, or definitions and interface specifications approximating your chosen language (if it has such constructs). Feel free to use examples and diagrams. You might also want to include a mock up of the actual UI (what information is shown and how it is to be arranged on the screen). This might also come in handy when designing the user interface for a human player.

Programming Task

Implement the Rule checker and Game Manager components. You might need to create a stub for a Player, based on your specification from Milestone 4. Note that the Game Manager implemented here will be incomplete (i.e., the level will have no adversaries), but will hopefully get us to a demo sooner.

For the rule checking infrastructure, implement the rules specified in Milestone 3.

A Game Manager should have the following responsibilities.

- Register between 1 and 4 players. The order of registration determines the order in which they take turns.
- Register an arbitrary number of adversaries (although we have no adversary components yet; also see the digression on adversaries below).
- Start and supervise a SNARL game. For simplicity, assume SNARL currently only has a single level and that the level can be passed to the Game Manager. If you already have infrastructure for multiple levels in place, keep it.

A SNARL play-through of a level breaks down into several turns. In each turn,

- the manager needs to update each player about any changes in the game state *as they happen*;
- when it's the player's turn, the game manager needs to request the player's move, check it for validity and, if valid, apply it to the game state; and
- if the player ends up on a tile containing an object or an adversary, the manager needs to apply the result of the interaction to the game state.

Scope: We will look for good code design, readability, unit tests, and whether we can find the functionality we asked for above in the code. We will also be looking for unit tests.

On Adversaries

We have left the specification of how we want Snarl to handle adversaries a little murky. Are they internal components? Do they connect via network? Are they plug-ins to be loaded from somewhere? The answer is, they can be all. In a final version of Snarl, we want to be able to accept adversary AIs via network to participate in the game, just like we do with human players. The idea is, since both are characters in a game, both players and adversaries can be treated in a similar way.
A problem that arises with relying on external adversaries is that, while running a game with only a single player is possible, if we have too few enemies, we might not really have an engaging game. Therefore, we want to leave the flexibility for the Game Manager to accept external adversaries, but also to fill in the blanks by creating local adversaries. We will design an API for adversaries in an upcoming milestone.

Testing Task

Create a test harness executable `testState`, which, given a state specification and a player with a destination, will output an updated state or an error message if the player cannot be placed in the destination.

Test Input

The input JSON is an array with the following format:

```
[(state), (name), (point)]
```

The array is to be read as follows: given the **(state)**, (try to) move the *player* with the given **(name)** to the position given by the **(point)**.

The following data definitions apply.

- A **(state)** is a JSON object with the following shape:

```
{
  "type": "state",
  "level": (level),
  "players": (actor-position-list),
  "adversaries": (actor-position-list),
  "exit-locked": (boolean)
}
```

Interpretation: A **(state)** collects the data needed to construct a SNARL game state with a level, player and adversary positions, and the status of the exit.

- A **(level)** is as specified in Milestone 4
- An **(actor-position-list)** is a list of **(actor-position)**. An **(actor-position)** is the following object:

```
{
  "type": (actor-type),
  "name": (string),
  "position": (point)
}
```

- An **(actor-type)** is one of:
  - "player"
  - "zombie"
  - "ghost"

Test Output

The test harness should output one of the following:

(1) If the given player exists in the input state, can be moved to the given position, and the position is not occupied by an exit or an adversary,

```
[ "Success", (state) ]
```

Where, **(state)** is an updated state. We are not checking rules, so the position can be arbitrarily removed from the original position, but the player has to land on a traversable tile (non-wall room tile or a hallway) not containing another player. If the player lands on a key, this should be reflected in the output state's exit status.

(2) If the player landed on an adversary,

```
[ "Success", "Player ", (name), " was ejected.", (state) ]
```

The player should be removed from the output state.

(3) If the player landed on an exit and the exit is unlocked,

```
[ "Success", "Player ", (name), " exited.", (state) ]
```

The player should be removed from the output state. If the exit is locked, (1) applies.

(4) If the player isn't part of the input state,

```
[ "Failure", "Player ", (name), " is not a part of the game." ]
```

(5) If the player's destination tile is not traversable (a wall, or outside of a room or hallway),

```
[ "Failure", "The destination position ", (point), " is invalid." ]
```

Other than the cases above, assume that the input JSON is always valid.

Note, it is not expected that the game state handles all the checks above itself. It should, however, support checking some of the above conditions (key, adversary, etc.) Feel free to include functionality you are implementing for the programming task above.

Scope: Your executable must run on the Khoury Linux VMs, accept JSON with the format given above on standard input and return results to standard output. We will be running our own tests and your team's tests through your testing harness. Ideally, your code from previous milestones shouldn't need to be modified extensively.