Project Part 4

CS 4500 Software Development

Due: Wednesday, March 3, 11:59pm

Submission: Create a release on GitHub tagged `p4`. Add a `p4-exec.zip` to the release containing:

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

Here is a sketch of the `p4-exec.zip` layout:

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

Submit a ZIP with the following on Handins:

1. A folder `Snarl/tests/Level` with all of your testing suite for the testing task
2. Any other files from your source tree that `testLevel` depends on
3. The files `game-manager.md` and `player.md` in `Snarl/Planning/` for the design task

Here is a sketch of the Handins ZIP layout:

```
Snarl
|-- Planning
|   |-- game-manager.md
|   `-- player.md
|-- tests
|   `-- Level
|         |-- testLevel.<ext>
|         |-- 1-in.json
|         ...
`-- src
    ...
```

Note: to avoid conflicts in your repository, you might want to rename `tests/Level` from Milestone 3 to `tests/Room` which now seems a better fitting name for that test harness.

Digression: Design for Change

Many of you have noticed that your instructors can't seem to get their doo-doo together and seem to disagree on basic details about the game. Does an adversary have to move on every turn? Does a level end if all players reach the exit or is one of them sufficient? You correctly caught us making contradictory statements. However, while it is fair to ask these questions, the actual answers to many of them do not (and should not) matter for your design and code.[a] Many of them will change and change again, and then change some more. At some point, "final" choices will be made and the finished product will embody those choices.
The bottom line is, weather, moods, requirements, design, software all change. You should strive to **design and program for change**. Whether an adversary *has to* move on every turn – that shouldn't take long to change. Do we require 1 to 4 or 2 to 6 players? Heck, there should be a constant for that somewhere! Does a level end when *any* or *all* players reach the exit? Better make that logic easy to find, comprehend, and, ultimately, modify, so when your instructors *finally* make up their minds, you're not caught by surprise.

[a]They do matter if we evaluate your code and claim it's not doing something we actually told you it shouldn't be doing. That's on us and a reason for a regrade.

Design Task

Design the interfaces for the Game Manager and Player components.

For now, the main task of the *Game Manager* is to accept players to the game and start a game with a single level, which will be provided. Players should provide a unique name when registering.

A *Player* component represents the interests of the human behind the keyboard in the game. A Player needs to receive updates from the Game Manager at appropriate moments. When it's the Player's turn, it needs to communicate the chosen action to the Game Manager.

Include definitions of any auxiliary data you might need to specify these interfaces. Consider any kind of information that is necessary for the Player or the Game Manager in their interaction.

Assume the following for now (also refer to the digression above):

a) A player can see at most 2 grid units away in any cardinal or diagonal direction. That is, a player (marked X) can, at any moment, see all the dots in the sketch below.

```
.....
.....
..X..
.....
.....
```

b) A player may choose to stay put.

c) Players only interact with items on the tile they've chosen as their destination.

d) A player knows its location in relation to the level's origin.

e) The order of interaction for a player is: interact with the enemy, interact with the key, and, finally, interact with the exit.

Scope: The purpose of this task is looking at how the Game Manager and Players should interact to start and run a single level. 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.

Vocabulary: Players and Players

We tend to use the term "player" in at least two meanings in discussions about SNARL:

i) The component representing a human player in the system. This component communicates the state of the game to the human and communicates the human's actions back to the game. In this milestone, we are designing this component.

ii) The player "avatar" inside a level, which has a position, some visual representation, etc. This one is typically part of the game state.

Testing Task

Create a test harness executable `testLevel`, which, given a level and a point in the level will output information about that point:

- whether it is traversable;
- whether the tile it references contains a key or an exit;
- if it is a hallway, the origins of the rooms it connects; and
- if it is a room, the origins of neighboring rooms, that is, the rooms that are one hallway removed from the current room

Test Input

The input JSON is of the following format

```
[(level), (point)]
```

where the following data definitions apply.

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

```
{
  "type": "level",
  "rooms": (room-list),
  "hallways": (hall-list),
  "objects": [ { "type": "key", "position": (point) },
               { "type": "exit", "position": (point) } ]
}
```

- A `(room-list)` is a JSON array of `(room)` as defined in Milestone 3
- A `(hall-list)` is a JSON array of `(hall)`
- A `(hall)` is a JSON object with the following shape:

```
{
  "type": "hallway",
  "from": (point),
  "to": (point),
  "waypoints": (point-list)
}
```

- A `(point)` and `(point-list)` are as defined in Milestone 3.

The `"from"` and `"to"` points specify *door tiles* on the boundaries of the respective rooms and so are considered a part of the room.

For this task, assume that the given level is valid and that it always contains exactly one key and one exit object.

Test Output

As output, the `testLevel` harness should output the following JSON object to standard output:

```
{
  "traversable": (boolean),
  "object": (maybe-object-type),
  "type": (room-or-hallway-or-void),
  "reachable": (point-list)
}
```

The data definitions below apply. The order of the fields in the outputted object does not matter.

- A `(boolean)` is one of
  - `true`
  - `false`

- A `(maybe-object-type)` is one of
  - `"key"`
  - `"exit"`
  - `null`
- A `(room-or-hallway-or-void)` is one of
  - `"room"`
  - `"hallway"`
  - `"void"`

The interpretation for the fields is as follows.

traversable `true` if the tile is traversable (door or "floor", i.e., type `1` or `2` in Milestone 3's testing task or a hallway), `false` otherwise.

object Type of the object if the tile contains a key or an exit. Otherwise `null`.

type The type of the current level segment. `"void"` is a grid position which is neither in a room, nor in a hallway.

reachable An array of room origins that are immediately reachable from the current room or hallway (if the given point is in one or the other). For a hallway, these are the rooms which it connects. For a room, these are the rooms which are immediate neighbors of the room, i.e., reachable by crossing exactly one hallway. If the point is not in a hallway or a room, the array should be empty. The order of the points in the array does not matter.

Note: we consider a hallway to be one tile wide. That means, even if you represent a hallway with a wall on either side somewhere, that wall is considered `"void"` for the purposes of this assignment.

Create a test suite for `testLevel` with at least 3 test file pairs. Your test files must be in pairs, `<n>-in.json` and `<n>-out.json`, where `<n>` is an integer greater than 0. `<n>-in.json` should only consist of the input and `<n>-out.json` should only consist of the output, both specified above.

Scope: Your executable must run on the Khoury Linux VMs and your tests must be correct. We will be running our own tests and your team's tests through your testing harness. Ideally, you should not need to modify your code from the previous milestones.

Example

In the example inputs, `%LEVEL%` is a *placeholder* that is to be replaced with the following JSON before feeding to `testLevel`.

```
{ "type": "level",
  "rooms": [ { "type": "room",
               "origin": [ 3, 1 ],
               "bounds": { "rows": 4, "columns": 4 },
               "layout": [ [ 0, 0, 2, 0 ],
                           [ 0, 1, 1, 0 ],
                           [ 0, 1, 1, 0 ],
                           [ 0, 2, 0, 0 ] ] },
             { "type": "room",
               "origin": [ 10, 5 ],
               "bounds": { "rows": 5, "columns": 5 },
               "layout": [ [ 0, 0, 0, 0, 0 ],
                           [ 0, 1, 1, 1, 0 ],
                           [ 2, 1, 1, 1, 0 ],
                           [ 0, 1, 1, 1, 0 ],
                           [ 0, 0, 0, 0, 0 ] ] },
             { "type": "room",
               "origin": [ 4, 14 ],
               "bounds": { "rows": 5, "columns": 5 },
               "layout": [ [ 0, 0, 2, 0, 0 ],
                           [ 0, 1, 1, 1, 0 ],
                           [ 0, 1, 1, 1, 0 ],
                           [ 0, 1, 1, 1, 0 ],
                           [ 0, 0, 0, 0, 0 ] ] } ],
  "objects": [ { "type": "key", "position": [ 4, 2 ] },
               { "type": "exit", "position": [ 7, 17 ] } ],
  "hallways": [ { "type": "hallway",
                  "from": [ 3, 3 ],
                  "to": [ 4, 16 ],
                  "waypoints": [ [ 1, 3 ], [ 1, 16 ] ] },
                { "type": "hallway",
                  "from": [ 6, 2 ],
                  "to": [ 12, 5 ],
                  "waypoints": [ [ 12, 2 ] ] } ]
}
```

Example Test Cases

1. Input:

```
[ %LEVEL%, [6, 10] ]
```

   Expected output:

```
{
  "traversable": false,
  "object": null,
  "type": "void",
  "reachable": []
}
```

2. Input:

```
[ %LEVEL%, [7, 17] ]
```

   Expected output:

```
{
  "traversable": true,
  "object": "exit",
  "type": "room",
  "reachable": [[3, 1]]
}
```

3. Input:

```
[ %LEVEL%, [12, 4] ]
```

   Expected output:

```
{
  "traversable": true,
  "object": null,
  "type": "hallway",
  "reachable": [[3, 1], [10, 5]]
}
```

Changes

02/28/21	• (Testing Task) Added the `"type"` field for hallways • (Testing Task) Clarified that the hallway endpoints are door tiles • (Delivery) Fixed test case filenames in examples • (Testing Task) Order of reachable rooms is not significant
03/01/21	• (Testing Task) Added the `"type"` field for levels • (Testing Task) Added example test cases