Project Milestone 3

CS 4500 Software Development

Due: Wednesday, February 24th, 9pm

Submission:

Create a release on GitHub tagged `p3`. Add a `p3-exec.zip` to the release containing:

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

Submit a ZIP with the following on Handins:

1. all new Source Code in `Snarl/src` for the programming task
2. a folder `Snarl/tests/Level` with all of your testing suite for the testing task
3. The `rulechecker.md` in `Snarl/Planning/` for the rule checker design task

Vocabulary

When we talk about *Game Manager*, we mean a component or a collection of components that collectively fulfill the responsibilities of running a game, managing players, enemies, etc.
When we say a tile is *traversable*, we mean a player can walk on it. Right now, these are tiles that are not walls.
When we say a point is *traversable*, we mean the tile found on that point in the level is itself traversable.

Design Task

Sitting through these Milestone 1 code walks, we found a flaw in our plan. In the plan, we required the Game Manager to perform too many responsibilities. The Game Manager has to run the game, manage the players and adversaries, and handle creating and loading levels all while hosting the game state (and possibly send that state out to players/adversaries). In the interest of keeping this one component lean, we will break this Game Manager component into sub-components. We will start with a Rule Checker component, a component/idea that came up in a few code walks.

The Rule Checker needs to validate the movements and interactions from players and adversaries as well as determine the end of a level versus the end of the game. It should also know when to reject invalid game states. Other functionality might be added later in the project if needed.

The movement rules for players are listed below.

- A player can move to any traversable tile up to 2 cardinal moves away from themselves. In other words, a player can move to a corner tile in 1 turn if it could reach that same tile in 2 moves using only cardinal moves. The chosen tile can be occupied by a key, exit, or an adversary, or nothing.

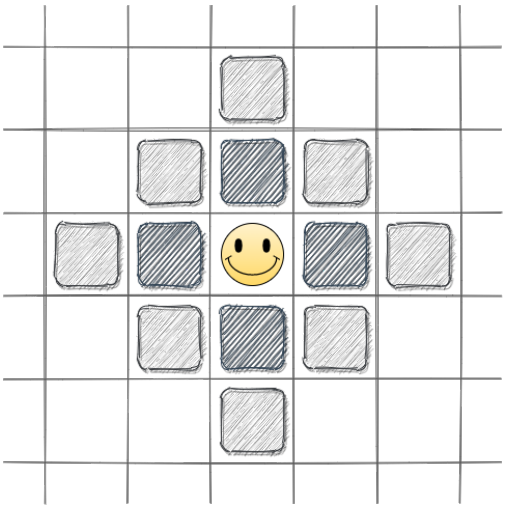

Figure 1: Tiles reachable in 1 (dark fill) or 2 (light fill) moves.

- The player can interact with keys and objects and adversaries (for self-elimination) but not other players.
- When the player moves, they interact with the object on the tile they choose and no other tile.

For details on the end of a level versus end of a game, refer to the Snarl Overview.

Your task is to design the rule checker's interface.

When designing, consider how your design could be extended to handle different kinds of adversaries with other movement abilities. For instance, a ghost type adversary could potentially ignore walls entirely.

Scope: The purpose of this task is looking at the required rules and detailing how the relevant components will interact with a rule checker.

Programming Task

Implement your game state design from Milestone 2. You will need the following functionality as well:

- Create an initial game state with a given level and some number of players (1 to 4) and some number of adversaries. To start, assume that players are placed in the top left-most room of the level, while adversaries are placed in the bottom right-most room of the level. Additionally, assume the designated rooms have enough traversable tiles to place players and adversaries.
- Create an intermediate game state with given player and adversary locations and the status of the level exit.
- Modify or create a new game state after a player or adversary moves or interacts with an object. The choice of "modifying the state" versus "creating a new state" will depend on your design.

Render the current game state. This means adding the ability to render a player and an adversary avatar onto the level.

If your design calls for another component (e.g. a Player or Adversary component), create a dummy/stub component for each missing component with no implemented behavior.

Scope: We will be looking for good code design, readability, and if we can find the functionality we asked for above in the code. We will also be looking for unit tests for the requested game state functionality. In particular, we do not expect unit tests for rendering in particular.

Testing Task

As part of testing, we will need a shared representation of levels so teams can share levels outside of the code base. We will split reading these representations into two parts across two milestones: one for rooms (the longest part), and one for hallways and levels all together.

Create a test harness executable `testRoom` that given a room and a point in the world will output a list of points in the room **1 cardinal move away that are traversable**.

Your harness will read JSON from STDIN and output JSON to STDOUT.

Test Input

The input is of the following format `[ (room), (point) ]`

where :

- a `(point)` is defined as `[Integer, Integer]` representing the location of a cell on a 2D grid as `[row, column]`. As an example `[0, 1]` is the cell at row 0 and column 1 in some 2D grid.

- a **(room)** is defined as follows

```
{ "type"   : "room",
  "origin" : (point),
  "bounds" : (boundary-data),
  "layout" : (tile-layout)
}
```

The `origin` is interpreted as ~~a Cartesian~~ the point of the top left corner of the room in a 2D grid representation of the level.~~The bounds are interpreted as width and then length. As an example for bounds, the fourth non-square layout in the Snarl Overview has a width of 7 and a length of 5.~~

- a **(boundary-data)** is defined as follows:

```
{ "rows" : Integer,
  "columns" : Integer
}
```

The two Integers in the object represent the number of rows in the layout and the number of columns in the layout respectively. As an example, the fourth non-square layout in the Snarl Overview has a 7 rows and 5 columns. The bounds would then be represented as follows:

```
{ "rows" : 7,
  "columns" : 5
}
```

- a **(tile-layout)** is an array of arrays (i.e., a 2D array) of Integers. The outer array ~~is `width` long and each inner array is `length` long~~ has "number-of-rows" arrays inside and each inner array has "number-of-columns" Integers. Each element of the inner array is one of 3 Integers:
  - `0` representing a wall (i.e. not traversable)
  - `1` representing a traversable non-door tile
  - `2` representing a door

Any other numbers are undefined and reserved for future use.

An example room is shown below:

```
{ "type" : "room",
  "origin" : [0, 1],
  "bounds" : { "rows" : 3,
               "columns" : 5 },
  "layout" : [ [0, 0, 2, 0, 0],
               [0, 1, 1, 1, 0],
               [0, 0, 2, 0, 0]
```

```
            ]
}
```

This represents a room somewhere inside of a large 2D grid, as shown below. We assume the top left cell is (0, 0). Unknown grid cells are rendered as ? since their contents are not of concern in this example.

```
? 0 0 2 0 0
? 0 1 1 1 0
? 0 0 2 0 0
? ? ? ? ? ?
```

Test Output

As output, the harness will print one of the following JSON array outputs to STDOUT:

- If the point is inside the room:

```
[ "Success: Traversable points from ", (point), " in room at ", (point), " are ",
  (point-list) ]
```

- If the point is outside of the room:

```
[ "Failure: Point ", (point), " is not in room at ", (point)]
```

Here, a `(point-list)` is a JSON array of `(point)`

Create a test suite for `testRoom` 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. The tests released for Warm-up 2 on Piazza are examples of such expected pairs.

This test harness must run on the Khoury Linux VMs.

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. You should not need to add code to your solutions from the previous milestone.

Pedagogy: The purpose of this task is to start thinking about integration testing. We will need to ensure all components can work together throughout the project.

For this task in particular, we are breaking level testing into 2 parts. This part focuses on reading JSON for the room interpretation.

By agreeing on the inputs and outputs ahead of time, we can create a large test suite for all teams to use after the testing task is complete. This is also a shared vocabulary for functionality questions and clarifications.

Example Tests

An example input for the harness (which can be in a file) follows

```
[ { "type" : "room",
    "origin" : [0, 1],
    "bounds" : { "rows" : 3,
                 "columns" : 5 },
    "layout" : [ [0, 0, 2, 0, 0],
                 [0, 1, 1, 1, 0],
                 [0, 0, 2, 0, 0]
               ]
  },
  [1, 3]
]
```

The expected output (which can also be placed in a file) follows

```
[ "Success: Traversable points from ", [1, 3], " in room at ", [0, 1] , " are ",
      [ [0, 3], [1, 2], [1, 4], [2, 3] ]
]
```

Notice the point (1, 3) is referring to the middle of the layout, indicating (1, 3) is not local to the room.

Another example input for the harness (which can be in a file) follows

```
[ { "type" : "room",
    "origin" : [0, 1],
    "bounds" : { "rows" : 3,
                 "columns" : 5 },
    "layout" : [ [0, 0, 2, 0, 0],
                 [0, 1, 1, 1, 0],
                 [0, 0, 2, 0, 0]
               ]
  },
  [5, 4]
]
```

The expected output (which can also be placed in a file) follows

```
[ "Failure: Point ", [5, 4] , " is not in room at ", [0, 1] ]
```