Comp2100 Pandemic Survival RPG Game

Team structure and roles

Anson (u7139215)

- Designing and implementing the fundamental structure of the game including parsing, NLP, menus and user interface, player movement and actions, items, fights, enemy attacks.
- Data structures for locations, levels and story nodes. Designing game levels.
- Automatic game testing
- Report writing

Phillip (u6943702)

- Storing and loading player and level data in JSON format (using bloom filter)
- Story and class ideas – enemies, items, locations, and their attributes
- Difficulty scaling of player and enemy damage and items

Nayoon (u7079736)

- Level design - writing the story and populating levels with content.
- Coming up with story ideas (enemies, items) implementing these elements in the levels.
- Hunting features – animals, items
- Writing unit tests for the game

Ahmed (u6700773)

- Little contribution (did not complete assigned work) - dropped out of the course

Project Summary

Description

Our game *Pandemic Survival* is an RPG game set in a post-apocalyptic world where a variant of covid-19 has infected most of the world and turned nearly everyone into mindless zombies. The player must set out to find a vaccine in order to save the world from the deadly virus.

Features of game engine

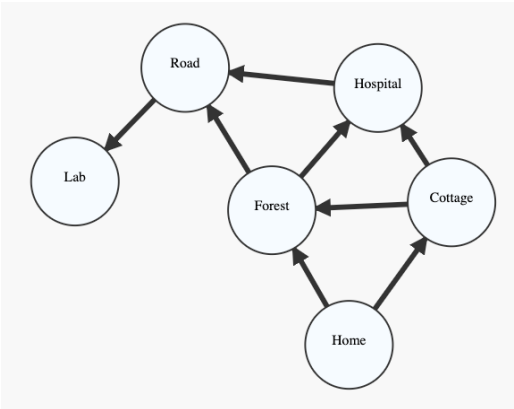
Saving and Loading Data

- Player can save and load their progress. This is implemented using JSON files which store player information. The files are clearly laid out and highly configurable. The game allows multiple players to save their game files based on their unique username.
- A bloom filter is used as a partial check for whether a username already exists or not. This increases the efficiency of the program when there are a large number of users saved in the database file.
- Levels and location data stored in JSON format and loaded into the game. The files are clearly laid out and highly configurable (i.e. levels can be designed or modified from the JSON file) as shown in the image on the right.
- A configuration file is used to store information such as zombie attack damage, player attack damage, medkit heal amount etc. for each level of difficulty. These values can be changed to adjust the balance of the game.
- The player can choose to save their progress when they complete a level. When the player dies, they can choose to load their past save

```
{
  "level": 0,
  "health": 100,
  "damage1": 60,
  "damage2": 35,
  "zombieApproachProb": 20,
  "zombieAttack1": 10,
  "zombieAttack2": 15,
  "zombieAttack3": 20,
  "psychopathAttack1": 10,
  "psychopathAttack2": 15,
  "animal": "BISON",
  "huntingKit": {
    "durability": 100,
    "quality": 100
  },
  "medKit": {
    "healValue": 50
  },
  "meat": {
    "hpAdd": 20
  },
  "energyConsumption": 2,
  "zombieHealth": 60,
  "psychoHealth": 60
},
```

Data Structures

- Locations and level nodes are stored in graph data structures with nodes and edges. Each node is assigned a unique ID for storage and linking purposes, and edges are defined by key value pairs between the IDs (in a map). Each location (level) has a story which the player can navigate by making certain decisions (selecting an option from the menu).
- For example, below is an illustration of our game map, implemented using a graph:



Design

- The overall design is very modular, meaning that pieces of code can be used for different purposes. For example, instead of manually printing out the options each time the player has to make a decision (e.g. choose option or location), we have used the Menu class to handle this.
- We have used inheritance to improve the modularity of our code. For example, the Zombie and Psychopath classes both inherit from the abstract class Enemy.
- We have also implemented an interface called menu item, for objects which can be displayed in menu format.
- Instead of creating a specific class for each location e.g. Cottage, Hospital, we have a general class called Location, which encapsulates the same functionality.
- We have used Death Exception and Win Exceptions to indicate when the player has died or beat the game. This is a simple, yet effective method for handling game termination.

Features of Game Tester

Automatic Game Tester:

- The game tester calculates and prints a report containing the number of paths, and the max/min/average number of nodes traversed for each level (location).
- From these values, it is easy to see if there is something wrong with the level map. For example, it was found from using the automatic tester that there were issues with certain locations since the number of paths, and the max/min/average values were under the expect value.
- The automatic game tester also detects loops. If the tester does not terminate, it means that there are some edges which connect to previous level nodes. This also ensures that the player never ends in an unplayable state, since the algorithm only terminates when the player reaches a level completion node.

```
Location: forest
Number of paths: 5
Average number of nodes traversed: 6.4
Max number of nodes traversed: 7
Min number of nodes traversed: 6
-------------------------------------

Location: road
Number of paths: 1
Average number of nodes traversed: 4.0
Max number of nodes traversed: 4
Min number of nodes traversed: 4
-------------------------------------

Location: cottage
Number of paths: 8
Average number of nodes traversed: 6.5
Max number of nodes traversed: 7
Min number of nodes traversed: 5
-------------------------------------
```

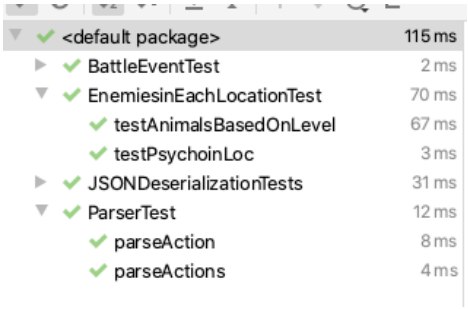
- Our aim for each level was to achieve a minimum number of nodes traversed of 5 for to ensure the user's experience would be enjoyable, and that the levels were not too short.
- The automatic game tester also provides these metrics for the whole game:

```
All levels:
Number of paths: 363
Average number of nodes traversed: 25.143251
Max number of nodes traversed: 29
Min number of nodes traversed: 13
-------------------------------------
```

- The automatic game tester can be found in the `AutomaticGameTester` runnable class.

Unit Tests

- We also implemented various unit tests for functions in our program. These include tests for enemies, battle events, the parser, and JSON serialisation and deserialization.



Advanced Features

Natural Language Processing

- Natural language processing has been implemented to parse complex user commands such as “use your medkit”, “look under the table” and “exit the game”. The user’s input string is tokenised and parsed based on the grammar defined below:

```
sentence = verbG1 objectG1 | verbG2 objectG2 | verbG3 objectG3 | verbG4
prepositionG1 objectG4
verbG1 = "exit" | "save"
verbG2 = "examine"
verbG3 = "use"
verbG4 = "look"
prepositionG1 = "above" | "below" | "inside" (get options from levelnode)
objectG1 = determinerG1 nounG1
objectG2 = determinerG2 nounG2
objectG3 = determinerG2 nounG3
objectG4 = determinerG2 nounG4
determinerG1 = "the"
determinerG2 = "your"
nounG1 = "game"
nounG2 = "items" | "stats" | "surroundings"
nounG3 = medkit (get items from player)
nounG4 = furniture
```

- Originally, there were some sentences which were not grammatically correct, for example: “get the exit”. This was solved by making the grammar more formal by creating additional groups e.g. G1 and G2 for determiners. Once parsed, the

sentences are evaluated. If there is an unknown word, or the sentence structure is incorrect, the evaluation will result in an error message indicating the problem to the user:

```
examine a
Unexpected: a
use
Incorrect sentence structure.
```

- The player can type “h” to receive a list of generated commands which have been processed using the parser.

```
Menu Items:
Type "0" to choose option: head to the bedroom
Type "1" to choose option: explore the living room
--------------------------------------
Addtional commands available to you:
    exit the game
    examine your items
    examine your stats
    examine your surroundings
    look under the table
--------------------------------------
```

Efficient Algorithms

- Dynamic programming was used to generate the automatic testing results efficiently. The number of nodes traversed for each level was calculated using a recursive algorithm and stored in a HashMap. When calculating the number of nodes traversed for the whole game, the previously calculated values were obtained from the HashMap, meaning that we did not have to recalculate the values.

Game Example Walkthrough and Explanation

```
┌──────── •✧✧• ────────┐
 -Pandemic Survival Game-
└──────── •✧✧• ────────┘
Options:
[0] - Start New Game
[1] - Load Existing Game
[2] - Exit
```

From the main menu, the player can choose to start a new game or load a game from a previous save.

```
Choose your difficulty:

Options:
[0] - Easy
[1] - Medium
[2] - Hard
[3] - Expert
```

The player can choose from 4 difficulty levels which scale the damage of enemies and usefulness of items.

```
-------------------------------------
Current location: home
Choose a place to go.
-------------------------------------

You are at home, wondering where you should go.
Choose a place to go.

Options:
[0] - cottage
[1] - forest
```

When the game begins, the player can choose a location to explore. The player will be able to choose a new location after completing a level.

```
Current location: cottage
You found a country-style cottage. Maybe you can steal some food? After
  all, you are very hungry and wanted to stock up more food and supplies.
--------------------------------------

Would you like to save your progress?(y/n)
```

The player can choose whether they want to save their progress. If the player dies or exits the game, they will be able to load their saved progress.

```
h
Help:
-------------------------------------
Menu Items:
Type "0" to choose option: explore the living room
Type "1" to choose option: head to the bedroom
-------------------------------------
Addtional commands available to you:
    exit the game
    examine your items
    examine your stats
    examine your surroundings
    look under the table
-------------------------------------
```

The player can type “h” or “help” for a list of commands.

```
Options:
[0] - explore the living room
[1] - head to the bedroom
0

You enter the living room and catch a glimpse of a young girl dozing off on
   the sofa in the small living room.
You see a something under the table in front of her.

Options:
[0] - wake her up
[1] - sneak past her
[2] - leave the room
```

The player can choose from the given options as shown in the example above through a numerical input. Their decisions will affect enemy encounters and rooms they will be able to enter later on.

```
examine your surroundings
You examine your surroundings.
--------------------------------------
You see:
    a bed
    a drawer
--------------------------------------
```

The player can see what objects (e.g. furniture) are around them using a text command.

```
look inside the drawer
--------------------------------------
You found a medkit inside the drawer.
medkit has been added to your inventory.
--------------------------------------
```

The player can examine these objects to find items.

```
A Psychopath (70hp) appears in front of you.
What would you like to do?
Options:
[0] - Head on attack (60dmg/-10hp)
[1] - Sneak attack (35dmg/-0hp)
0
You choose to use a head on attack.
Psychopath 10hp(-60hp)

Psychopath punches you in the stomach. -30hp
Anson: 60hp
```

Some decisions will lead to enemy encounters. The player has multiple options for attacking. In this example, a head on attack will deal 60 damage, but the player will also take

10 damage. A sneak attack will deal 35 damage to the enemy, but no self-damage. The player will have to decide whether they want to maximise their damage or conserve their health points. Enemy Attack damage is based on the difficulty (and has some degree of randomisation)

```
What would you like to do?
Options:
[0] - Head on attack (60dmg/-7hp)
[1] - Sneak attack (35dmg/-0hp)
use your medkit
You have used a medkit. +20 Hp
```

If your hp is low, you can use a medkit to heal yourself; this is useful during fights.

```
Having been hungry, you decided to hunt after confirming that you didn't
 have enough food.

Hunt a BISON (20hp) in front of you.
What would you like to do?
Options:
[0] - Headshot (30dmg/-10hp)
[1] - Attack its leg (17dmg/-0hp)
```

The player can also hunt different types of animals for meat (deer, rabbit, trout, bison).

Application of Course Content

- We have applied tokenisation concepts explained in the lectures and used these tokens to build parse trees based on a predefined grammar (set of production rules)
- Player can save and load their progress from JSON files. Location and level data are stored in JSON format and loaded into the game.
- A bloom filter data structure is used to efficiently check if username exists when creating a new profile or loading a profile.
- We have applied the concepts explained in the lectures to implement unit tests.
- We have applied the ideas of design by contract when designing our code. For example, the automatic game tester relies on the assumption that the first and last levels have only 1 node. The remove function in the `PlayerJSON` file expects that the player to be removed actually exists.
- When coming up with our first design, we used class diagrams to plan out the structure of the game.
- We have applied ideas discussed in the lectures such as inheritance (including abstract and interface classes) and polymorphism. Polymorphism was useful when designing the token parser

Team meeting minutes

- Our team set up weekly meetings where we discussed what had been completed and what needed to be completed by the following meeting.
- More detailed information can be found in a file named ‘team_meeting_minutes.pdf’ inside the root directory.

Conclusion

I believe we have successfully created a high-quality, creative game which meets all the requirements set out in the assignment specification sheet.