Project Report

FORC - PA5 – Open Assignment – Rogue

danielk21@ru.is - markusf21@ru.is - laufeyf20@ru.is

Project Description

For our project, we aimed to create a terminal-based game like the classic dungeon crawler Rogue from 1980. Our version of the game is a simple dungeon crawler with no end objective. The player navigates a series of dungeon floors, trying to get through as many as he can. The dungeon is built up by combining a random templates of dungeon rooms. The player can traverse the dungeon using the arrow keys. The player can use items they find, and attack monsters they encounter during their adventures in the dungeon. Each dungeon delve will be its own adventure, meaning there is no persistence between runs. Instead of normal game ricks, which operate based on time, our game ticks will be triggered by movement. Every time the player moves, the monsters will have an opportunity to do so as well. The player will have to be careful around the monsters, as they attack quickly. It can be deadly to stay around them for too long.

Fulfilled Items from the Project Proposal

Item number one: Make entire new program with similar methods to some previous assignment.

We for example read from a file like in assignments 2 and 4, utilize classes and data types that we have used in previous assignments.
Originally estimated to 60 points, adjusted to 25.

Item number two: Make program take simple decisions.

The input handler makes simple decisions. Whenever the player presses an arrow key the players character moves to the relevant square as long as there is nothing in the way, i.e. a wall or monster, if there is a monster where he tries to move, he instead attacks it. Whenever the player moves the monsters in the game get a turn.
Originally estimated to 20 points, adjusted to 15.

Item number three: Random dungeon generation

A random dungeon is generated whenever the player starts the game or reaches a new floor. When the game is started room templates are read from a file and stored. Then when a new dungeon is generated, it chooses random rooms and arranges them into the dungeon matrix. Lastly it adds tunnels between the rooms so that the player can navigate between them.
Originally estimated to 40 points, adjusted to 25.

Item number four: Simple monster AI

Every time the player moves their character the monsters on the field take their action. If the player is within a certain radius from the monsters they will attempt follow, if the player is in an adjacent index the monster will attack, the monster can attack and move on their turn and it can also attack diagonally. If the player is far away the monster will move in a random direction.
Originally estimated to 40 points, adjusted to 20.

Other Implemented Items

We added a few features not mentioned in our list of requirements, although mentioned as possible features in the project description. These items are the following:

Boost items

These items can be found in various locations throughout the dungeon floors. Some rooms have predefined locations for items. Included items are, a Strength booster marked with an ‘S’ on the map, and a Health booster marked with an ‘H’ on the map. The Strength boost adds one strength point to the player, and the Health boost adds 2 health points.

This feature is worth 5 points.

Fighting

Fighting between the player and monsters is possible in the game. The player attacks by walking into the monster's square, but instead of moving on top of him, the monster takes damage equal to the players strength. The monsters attack a bit differently, while the player can only move or attack, the monsters can do both in the same round, and they can attack diagonally as well as to the sides. This makes them vicious to be around for too long.

This feature is worth 10 points.

Problems we ran into

We had issues trying to make pathfinding for the monster AI. The first though we had to solve the pathfinding for the ai was to use Dijkstra to calculate a path. We then implemented it, and tested it, but it was hard to make it work with obstacles. We then tried DFS and BFS, but both were way to slow for the size of our matrix. We then tried to implement A*, but never got that to work fully. Realizing our pathfinding algorithm was running way to slow, and would result in massive input delay for the player since every monster would need to run the algorithm to find a path, we started thinking outside the box (or inside the array? 😉). To try and reduce the time needed to find a path, we decided to try and limit the monsters vision. We did this so we could send our algorithm a subarray of the whole array to find a path in. This worked much better, but having to create the subarray, and then find paths for it still took a bit too long. Not wanting to limit the monsters vision too much, since that would make the game less fun, we decided to implement a more simple AI. The AI we settled for used xdiff and ydiff (in relation to the player) to decide its next action. This is simple math, which makes it incredibly fast to run, the downside being the AI won't always take an efficient path.

We had issues with taking input from the user. First, we tried using *getch*() from *conio*, later realizing that *conio* is not supported by all systems and was deemed deprecated a long time ago. Our solution to this was using *ncurses* instead. *Ncurses* allows us to use a different version of *getch*() supported by all systems after installing the *ncurses* package.

We had a problem with the screen flickering. Originally, we printed each line on the screen using *cout*, this caused flickering and was generally slow. After making the switch to *ncurses*, *cout* was no longer an option as it is not supported within a *ncurses* screen. Instead, we tried using *printw*() to re render the whole screen. This was also slow and caused flickering. So we decided to try to rerender only the squared that needed to be changed. Using *mvaddch*() we could do this rather easily, resolving the flickering problem and making rendering the screen significantly faster.

Interesting Solutions

We implemented the random dungeon builder by taking random rooms from our list of rooms. They all have a standard size and format so that they all fit. Then we add them to our dungeon of size 2x2 rooms and copy the rooms in with the appropriate offset so that they all go into their own quadrant. Then we draw the tunnels by starting in the middle of the first quadrant and iterating left, when we hit a wall we start adding the tunnel walls until we hit another wall, thus having connected the two rooms together. Then we use the room height as an offset so we can make the tunnel between the lower two rooms. Then we do the same for the vertical tunnels.

Project Worth

Description	Points
Make entire new program with similar methods to some previous assignment.	25
Make program take simple decisions	15
Random dungeon generation	25
Simple monster AI	20
Boost items	5
Fighting	10