CS 32 – UCLA – Project 3 (MiniRogue) Report

I have written a moveGoblin function in the Actors.cpp file, which uses recursive findPath and traceBack function to theoretically move the goblin in the shortest path to the player. **IT SHOULD BE NOTED THAT THIS FUNCTION DOESN’T FULLY WORK AND SO I USE THE SAME MOVE FOR THE OTHER MONSTERS FOR THE GOBLIN WITH ITS SMELLDISTANCE.**

Regardless, the report will highlight the idea behind the recursive function written, and describe the algorithm that should work, but possibly doesn’t due to some syntax error.

1. **Class Design**

The design behind my implementation of the MiniRogue game is based off the following graph:

MAIN

GAME

DUNGEON

ROOMS

ACTORS

GAMEOBJECTS

PLAYER

MONSTERS

The game contains a pointer to a Dungeon. The Dungeon has a pointer to a player; it also has a vector of pointers to actors that are all monsters, and a vector of pointers to game objects. Throughout the execution we follow the dungeon pointer to get to the player or the monsters or the objects. The actors also have a pointer to the dungeon and a game object, so they know which dungeon to belong to and what weapon they wield. The game objects do not point to anything specifically. The Rooms belongs in the Dungeon.h, and is used in the dungeon, to hold a vector of rooms.

The players and respective monsters all derive from actors. Most of the heavy lifting of the actors is handled by the base class Actors. It has all of the positions, weapons and functions to move, move Monsters and attack. Upon destruction it destroys the weapon the actor was wielding. The actor class also has all the information such as the hit points, strength, dex, armour, sleep etc. and has accessors and mutators for them. The player has an inventory (that allows it to wield weapons, read scrolls and employ their effects and view the inventory), cheat (for purposes of debugging), heal, pickup and win that separates it from the Actor base class. The monsters have overrided dropItems function that is different for all the monsters, and the dragon and goblin have different ways of moving. The Dragon overrides the moveMonsters function to make it stationary, and attack when a player is next to it, while the goblin has the function described in the introduction of this report to enable it to move recursively (however this does not work and exists in the code to show the thought process behind my recursive idea).

The different weapons and scrolls all inherit from the GameObjects base class that has separate constructors for weapons and scrolls. I chose not to create a separate class for weapons and scrolls, because other than their name, actions and symbol, they are pretty much the same. The gameObjects have an action name, name, symbol, position, and type. Along with this they all have bonus dex and bonus damage they provide to the wielder. The specifics (i.e. their effects) of each scroll are actually employed within the inventory call for a player, thus removing this need for having separate classes for weapons and scrolls. Obviously the main difference between each derived object is its values for bonuses.

The Dungeon stands alone and doesn’t inherit from anything, but does most of the heavy lifting of the entire game, creating rooms and connections, creating levels, adding monsters, objects and the player, generating the display so on and so forth.

1. **Non trivial algorithms/programs with pseudocode**

I believe the following algorithms to be non trivial, while any other code is explained in detail within the comments itself:

1. **moveMonsters**
2. **createLevel (which employs making the dungeon maze)**
3. **attack**
4. **gameplay**
5. **the functions for recursive Goblin move**

moveMonsters(target location, smell range)

{

distanceToMove is the difference between our current position and end position (sum of the rows and cols)

if the distanceToMove exceeds the smell range, then return false

Do the following for North, South, East and West

if the moving in direction has no walls or other monsters

if the newdistance after moving in direction is less than the current distanceToMove

if player sits at that position

attack and return true

else update location and return true

return false

}

attack(opponent)

{

hit is false and attack string is empty

calculate the attackerPoints and the opponentsPoints using the formula in the spec

if the random value within the attackerpoints is greater than or equal to the random value within the opponentsPoints

the hit is true

if ourself is the player or the opponent is the player //Doesn't involve monsters vs monsters

if hit is true

calculate the damage points based on the formula in the spec

decrease the opponents hit points by this value

if the attack doesnt kill

set the attack string to represent the appropriate value

Check if the opponent or the attacker has equipped magic fangs of sleep

apply the possibility of sleeping, if it sleeps change attack string to represent this change

else print out the normal attack string taking in magic fangs

else if the attack kills

set the attack string to represent the final blow

set the opponent to dead

and remove the monster //We dont remove the player, because thats a special case as it exits the game

else if the hit is false

set the attack string to represent the miss

}

createDungeonMaze(number of rooms)

{

for the rooms

set a random start location, with random rowscales and colscales

if our start location surpasses the halfway line for the rows or columns

reverse the respective scales to negative, so when we make a room, its flipped

//For the four different cases of the values the scales can take get their respecitve walls

if our scales our positive

record the relevant leftwall, rightwall, topwall, bottomwall

create a room

add it to the vector of rooms

for the current number of rooms in the vector

check if any intersect by comparing the walls with respect to each other

if the intersection is true, remove this room and try again

else if no intersection

make the room by filling in the spaces as per the start location and scales

sort the rooms in the vector from left most centercolumn to rightmost

for the rooms in the vector

connect the centers of every room to the room right of it

}

createLevel(level number and monster chooser)

{

//Special cases are levels 0 and level 4

if level is > 0

delete the old vector of monsters, objects and clear out the old rooms

createDungeonMaze(random number of rooms)

choose random start location

if level == 0

addPlayer to start location until the location is valid

else

teleport player to new level and location until it is valid

add a random number of objects in each level between 2 to 3 to valid locations

add a random number of monsters in each level based on the level and monsterchooser valid locations

if level is between 0 and 3 inclusive

add stair case

if level is 4

add statue

}

play()

{

makelevel 0 and display it

create a getcharacter variable

while the variable is not 'q', we havent won or died

if we type h,j,k,l or any arrowkeys

move the player in said direction

try heal

move the monsters

display updated dungeon

if we died then exit

if we type i,w,r

get the players inventory and display it

if we type c

cheat

moveMonsters

display updated dungeon

if we type g

if we are above an object that can be picked up

if we are above '&'

we have picked up the golden idol, and so we win

and exit

else pick up the object and set its pickup string

move the monsters

display the updated dungeon with the pickupstring

if we type >

if we are above >

increment the level

make the new level

display the new level

get the new character input

if we pressed q or died

display the relevant message until q is pressed

}

**GOBLIN MOVE:**

**The idea behind my recursive move for the Goblin involves Lee’s algorithm. We create a secondary grid of integers with walls and other monsters being represented as -1’s, while blank spaces are represented by 1000, and the monsters position is 0. We then create a fillPath algorithm that will fill this grid with integers moving radially out wards from the 0 value, incrementing by one each time. Eg: from the 0 position if we have 4, 1000’s then we replace them with 1’s and from those 1’s we go outwards writing over 1000’s by 2’s, in such a way that when we look at the player, if he is at a position where the secondary grid value is goblinSmellDistance or less, then you traceback from the player to the goblin, pushing the coordinates on the stack, and this will give you the shortest path. We thus move the goblin to the top of the stack’s direction, if we end up at the player’s position, we attack. Clear the stack and start over. This obviously becomes memory intensive, but should serve as a valid algorithm to recursively move following the shortest path.**

**The path would like something like. Therefore we can see that after filling the paths, tracing back from end to start in consequential order will always lead back to the start in the shortest way possible.**

0

2

1

3

3

4

4

5

6

5

7

7

8

6

7

8

-1

-1

-1

-1

Start

End

setSecondary()

{

for the rows

for the cols

if it is a wall or monster

set this point as -1

else

set it as 1000

set this->location as 0

}

fillPath(current position, start position)

{

//Base case

for the rows

for the cols

if the grid has been filled with values that arent -1

decide is set to true

else

decide is false

if decide is true then return

//For the 4 directions, North, South, West and East

if its a valid move in that direction

if the value in the secondary grid at the position we wish to move to is greater than our current position's value +1

set it to our current position's value +1

fillPath(new position, start position)

}

pathExists(player's position, smell)

{

if the secondary grid's value at the player's position is greater than the smell range

return false

else return true

}

traceBack(start position, end position)

{

while we dont hit the 0, i.e. the monsters position

//For the four directions North, South, West, East

if the value in that direction in the secondary grid is your value -1

push the coordinates onto our direction stack

}

moveGoblin(Goblin's start position, Players position, smell)

{

fillPath(start,start)

if pathExists(players position, smell)

traceback(players position, goblins position)

else return

if the goblins position is the players position

attack player

else

move in to the coordinates of the top of our stack and pop them off

clear the direction stack

}