**Goblin Movement**

In project three, the goblin movement function delegates all its work to three helper functions to determine the optimal move for the goblin [if there exists one]. These three helper functions are declared in the Actor.h file and are implemented in the Actor.cpp file.

bool Goblin::pathExists(int maze[18][70], int sr, int sc, int er, int ec, int dist);

int Goblin::shortestPath(int maze[18][70], int er, int ec, int sr, int sc);

void Goblin::resetMaze(int maze[18][70]);

Out of the three helper functions, shortestPath is the helper function that utilizes recursion to determine the shortest grid distance from the player to the goblin. The pathExists helper function is used to determine whether a path is possible [between the player and the goblin] via the use of a queue (exactly in the manner of homework two, question three), and the resetMaze function is used to reset the test maze (as arrays are passed by reference).

**Program Design**

N.B. As dynamic cast was not used in the entirety of the project, base classes will contain some virtual functions that are not used (this is intended to see full functionality of the base and all of its derived classes). As a result, these unused functions do not have parameter names in their implementations (as these are virtual functions with no intended use for the base class, but not all derived classes can use them).

Item: This is an abstract base class from which the Weapon, Scroll and SpecialItem classes are derived from. It holds information such as the Item’s coordinates, its owner [if any], the Dungeon it belongs to, its statistic boosts [if any] and its name and type (which are used for the display, action and result strings).

Weapon: This is a class derived from Item. In addition to the base functionality, Weapons can be wielded from the inventory.

Scroll: This is a class derived from Item. In addition to the base functionality, Scrolls can be read to provide a boost to the Player’s statistics.

SpecialItem: This is a class derived from Item. It has no additional functionality beyond the base class, as they are special types of Items (i.e. the stairway and golden idol). Triggering the activation of these objects will result in either a new level or the end game.

Actor: This is an abstract base class from which the Player, Bogeyman, Snakewoman, Goblin and Dragon classes are derived from. It holds information such as the Actor’s coordinates, its statistics, its type (used for the display, action and result strings) and the Dungeon it belongs to. All Actors can move, attack and have their statistics changed, amongst other actions.

Player: This is a class derived from Actor. In addition to the base functionality, Player maintains a vector inventory of its Items as well as the vector index of the currently wielded Weapon. The Player can also read Scrolls, wield Weapons, pick up Items, and display their inventory.

Goblin: This is a class derived from Actor. In addition to the base functionality, Goblin maintains an Item pointer to its wielded weapon, as well as its smell distance parameter. Goblins utilize a special smell algorithm to move appropriately (see first section of report).

Other Monsters: These are all classes derived from Actor. In addition to the base functionality, Bogeyman, Snakewoman and Dragon all maintain an Item pointer to its wielded weapon.

Dungeon: This is a standalone class that holds crucial information about the Game state, such as the status of each cell, the Goblin’s smell distance parameter, the current level, the total number of monsters and maintains two vectors, one to keep track of the amount of available Items and another to keep track of the floor’s monsters. In addition, Dungeon holds different output strings, which are set by the Actor actions and from the action strings of the Items. The Dungeon can add/remove Items and Actors, generate new floor layouts, set the status of different cells and the output strings.

Game: This is a standalone class that holds the Dungeon. The Game can run, execute and take Actor turns.

**Non-Trivial Algorithm Pseudocode**

bool Player::pickUp(int r, int c, bool gflag)

Retrieve Player’s inventory size

If function called with ‘g’ and inventory size under 26:

Set output string depending on Item name

If Item is special, activate new level/end game

Set Item owner to be the Player, and add to the inventory

Remove Item from Dungeon

Else if function was called with ‘>’:

If Item is stairway, generate a new level

Else:

If Item is golden idol, end game

bool Goblin::pathExists(int maze[18][70], int sr, int sc, int er, int ec, int dist)

Push the starting coordinate onto the coordinate queues

Update the maze[sr][sc] to indicate algorithm has encountered the cell

While the queue is not empty:

Pop the front coordinate off the queue

If front coordinate is the end coordinate, return true

If South is available and not visited, update maze and push to queue

If West is available and not visited, update maze and push to queue

If North is available and not visited, update maze and push to queue

If East is available and not visited, update maze and push to queue

There was no solution, so return false

int Goblin::shortestPath(int maze[18][70], int er, int ec, int sr, int sc)

Set the initial distance traversed to be zero

If the current coordinate does not equal the target coordinate:

Retrieve the North, South, East, West cell status

If any cell is a wall, invalidate that cell

If North is available and contains the minimum marked cell, decrement er

If South is available and contains the minimum marked cell, increment er

If West is available and contains the minimum marked cell, decrement ec

If East is available and contains the minimum marked cell, increment ec

Increment the distance

Recursively call shortestPath to determine the shortest grid distance

Return the final distance

bool Bogeyman::move(int drow, int dcol) // Same method as Snakewoman’s move function

Obtain the Player’s coordinates

If the Manhattan Distance is greater than five, return and do nothing

Else if the Manhattan Distance is one, call attack and return

Else:

If North is available and decreases the Manhattan Distance, move North

Else if South is available and decreases the Manhattan Distance, move South

Else if East is available and decreases the Manhattan Distance, move East

Else if West is available and decreases the Manhattan Distance, move West

void Dungeon::generateLayout(int level)

Generate a random number of rooms, minimum of four

Initialize two vectors to keep track of the rooms’ corner points

For all the Dungeon rows and columns, set each cell to be a wall

While the number of rooms is not zero:

Generate top left corner and room dimensions

Generate corridor starting coordinates, and push these onto the vectors

For the width and height of the room, “eat out” the room cells

Decrement the number of rooms remaining

Generate a random number of items, either two or three

If the last level, add the golden idol, otherwise add a stairway

For all elements in the two vectors initialized above:

While the starting coordinate does not equal the next coordinate:

Eat out the cells if they are walls only

Generate a random number of monsters

**Non-Trivial Algorithm Design**

Goblin’s Shortest Path: The idea here is to work from the target cell backwards to the starting cell, by travelling along the marked cells with the smallest value. This is possible because in the path exists function, the use of a queue will mark all neighboring cells with the same distance level (as queues search by breadth, not depth), meaning that when we work backwards in this function, choosing the minimum value cell guarantees that we are traversing along the shortest grid path (as opposed to zig-zagging through the grid). This recursive method for determining shortest grid distance would not be possible if our path exists function solved the maze with a stack container (as stacks search by depth, not breadth). For further explanation, check out the pseudocode (see second section of report) and the commented explanation inside Actor.cpp.

Dungeon’s Room Layout: The idea here is to systematically work out the layout in small steps. First, begin by generating the number of rooms (from a minimum of four rooms to a maximum of eight rooms for level four). Initially, the entire floor will be all walls, but as we generate our rooms, we will “eat out” these wall cells and make them empty, by defining the room’s top left coordinate and its dimensions (width and height). In addition, we will randomly choose a point offset from the top left corner to start “eating out” our corridor, so that we do not generate isolated rooms. By having a vector store the corridor start points, we can loop through the vector and eat out corridors that connect the corridor start points, stopping one short of the end. This will ensure that rooms are connected. For further explanation, check out the pseudocode (see second section of report) and the comments inside Dungeon.cpp.

**Miscellaneous**

As of project turn in, all features are implemented, with no known bugs or serious inefficiencies. For reference, clarification or explanations on any additional algorithms you might deem non-trivial, check out the pseudocode and descriptions provided (see second and third sections of the report), as well as my guided comments (which explain my thought process through the code) and actual implementation of the functions in their respective .cpp files. Output strings, capitalization and grammar syntax for the display were chosen at my own discretion, as the specification and executable differed somewhat with the string presentation (e.g. the specification states: “Your armor glows blue”, but the executable states: “Your armor glows sliver” (this is NOT a typo)). The cheat command in my project aligns itself more closely with the executable, setting the Player’s health to 50 and its strength to nine, and like the executable, can only be called once per game. The other notable comment is that any tile that somehow displays the character ‘E’ signals that the Dungeon thought that the cell contained an Item, but no Item object was ever found at that cell.