**Computer Science 32: Project 3  
Report**

1. Recursive Goblin Function

The recursive goblin function is named as below and is located within the Dungeon.cpp file.

int goblinSearch(char map[18][70], Goblin\* monster, int r, int c, queue<Coord> path, int shortest);

1. Program Design

***Actor Class***

The Actor virtual class had five derived subclasses. Each Actor object inherited the following private member variables and public functions in addition to simple accessors and mutators.

|  |  |
| --- | --- |
| string m\_name; | Name of Actor |
| Coord\* position; | Coordinate of actor position in dungeon |
| int m\_hp; | Current hp of actor |
| int maxHP; | Maximum hp of actor |
| Weapon\* m\_weapon; | Weapon that actor is currently wielding |
| int m\_armorPts; | Armor stat of actor |
| int m\_strengthPts; | Strength stat of actor |
| int m\_dexterity; | Dexterity stat of actor |
| int m\_sleepTime; | Sleep status of actor in number of turns |

**Significant Actor Public Member Functions:**

**virtual string attack(Actor\* foe);**

This function is the generic attack function utilized by all actors in the game. It takes in a pointer to an Actor which allows for each character in the dungeon to update the appropriate hit point and sleep time statistics.

Pseudocode:

*Calculate attack points with weapon bonus and player stats*

*Calculate foe’s defender points with armor and dexterity*

*calculate weapon damage points*

*make empty display string*

*if by random chance (1-attack points) attack is > defendant points (1-defensepts)*

*set foe’s hp to take damage*

*if foe is dead*

*update display string*

*if weapon is magic fangs*

*by random chance, determine if foe falls asleep*

*update display string*

*otherwise, attack missed*

*update display string*

*return display string*

**virtual void move(int r, int c);**

This function is a generic move function utilized by all actors in the game. It is used in implementation of the gameplay in order to move characters to the appropriate spot on the board. Position moved (as in Goblin movement, if the spot is open, etc.) is determined in Dungeon.

***Player Subclass:***

In addition to all the inherited variables and functions, the Player subclass has a vector of GameObject\* which represents the player’s inventory. The Player subclass also has accessor and mutators to add and remove items from the inventory, updating and deleting memory appropriately in order to prevent memory leaks.

***Bogeyman, Snakewoman, and Goblin Subclasses:***

In addition to all the inherited variables and functions in the Actor subclass, the monster subclasses implement no new additional variables. However, each has its own specifically defined weapon and character stats.

***GameObject Class***

The GameObject class is an abstract virtual class that defines all weapons and scrolls dropped and wielded in the game by various characters. One cannot make a GameObject object. Each GameObject contains the following private member variables, public member functions, and simple accessors and mutators. They also have a constructor and destructor.

|  |  |
| --- | --- |
| string m\_name; | Name of actor |
| Coord\* obj\_Pos; | Pointer to the position of the object. |

**virtual string action() = 0;**

This function is what makes the GameObject class abstract. Each GameObject item subclass returns a specific string describing the weapon or scroll action displayed on the dungeon screen during gameplay.

***Weapon Subclass***

The Weapon class is a derived abstract virtual class that defines weapons even more. In addition to the private member variables and public member functions inherited from the GameObject class, this class has the following two additional private member variables. The action() member function is still a pure virtual function within the Weapon class, meaning that the class is still an abstract virtual class.

|  |  |
| --- | --- |
| int m\_dexBonus; | Dexterity bonus from wielding the weapon |
| int damage; | Damage taken from being hit by weapon |

Each of the weapon types are a specific derived virtual subclass. Each weapon has its own action() function that specifies its action string. This action string will be used in formulating the outputted game update following usage of that specific weapon.

***Scroll Type Derived Subclasses***

Each scroll type has its own derived subclass. It inherits all the public members and private functions from the GameObject abstract virtual class but does not define any of its own unique members or functions. Each scroll subclass has a specific implementation for the action() member function, returning the appropriate string for the scroll action.

**Dungeon Class**

The Dungeon class is responsible for implementing all gameplay by updating Actor, object, and player status. Dungeon includes the following private variables:

|  |  |
| --- | --- |
| Player\* m\_Player; | Pointer to Player object, which tracks the player |
| vector<GameObject\*> m\_FloorItems; | Pointer to vector of GameObjects, which keeps track of all items spawned on the floor of a dungeon level |
| vector<Actor\*> m\_Monsters; | Pointer to vector of Actors , which keeps track of monsters spawned on each dungeon level |
| bool win; | Keeps track of if player has won or not |
| int goblinSmell; | Smell range of goblins (for recursive function) |
| int num\_monsters; | Number of monsters spawned per level |
| int num\_objects; | Number of objects spawned per level |
| char maze[18][70]; | 2D array that contains maze design of level |
| Coord\* m\_idol; | Pointer to a Coord holding the position of the idol |
| Coord\* m\_staircase; | Pointer to a Coord holding the position of the staircase for the current level |
| queue<Coord> steps; | Holds a queue used in the implementation of the goblin search function. |

The Dungeon uses all the Actor and GameObject classes in its implementation and is responsible for keeping track of character movements throughout gameplay. These member variables are used in the implementation of Dungeon ‘s member functions described below.

**Constructor/Destructor**

**Constructor:**

In the constructor, I initialize all values with either nullptr (if it is a pointer to a currently unutilized part of the game) or with the values needed for the 0 level of the dungeon. Pseudocode:

*Set player and idol to nullptr  
Call makeMaze() to initialize the first maze for level 0  
Add GameObjects to level 0  
Add Monsters to level 0  
Add staircase to level*

**Destructor:**

In the constructor, I deallocate all dynamic memory as shown below

Pseudocode:

*Call delete on pointer to player, idol, staircase  
Free GameObject items  
Free Monsters  
Free Rooms*

**Game Mechanics and Set-up**

**bool makeRoom();**

This function adds a room to the maze and is implemented in the algorithm for making a dungeon (described in part 4 of this report).

1. Randomly generate a point that corresponds to an open spot on the board.
2. Randomly generate dimensions of a room.
3. Check that the generated room fits within the dimensions of the board.
4. Check that the generated room is open on the board and does not overlap with existing rooms.
5. If a room is successfully made, return true. Otherwise, return false.

Pseudocode:

*Randomly choose values for start coordinate of room and dimensions  
Check that the generated room falls within grid dimensions  
Check that rooms are not overlapping with existing rooms (by checking that no space is equal to ‘ ‘)*

*if it is, do not make room  
if room fits and does not overlap*

*iterate over row/column and make room*

**void makeMaze();**

This function clears the map and creates a new dungeon for each corresponding level by calling the makeRoom(); function. It then follows an algorithm described in part 4 of this report to generate new rooms and corridors for each level.

Pseudocode:

*Clear previous level’s room*

*Randomly choose number of rooms for level*

*Add rooms by calling makeRoom()*

*Add corridors*

*...see part 4 for algorithm to create corridors...*

**void setLevel(int level);**

This function is responsible for setting up a new level by calling the needed functions to remove and delete objects and monsters from the previous level and add new objects and monsters for the new level. This function calls makeMaze(), addMonster(), and addObject() after using iterators and loops to delete the previous level’s vector of monsters and objects. Then it adds a staircase or the golden idol based on the level. setLevel(int level) is called after the player inputs the ‘>’ character in the Game class to move to the next level of the dungeon in the game.

Pseudocode:

*Update to new level*

*Delete last level’s GameObjects, Monsters, and Room objects*

*Make new maze and rooms*

*randomly generate new start position for player*

*Add new GameObjects and Monsters*

*Add staircase OR golden idol by checking the current dungeon level*

**void addPlayer();**

This function adds a new player to the board at the start of the game. The item m\_Player is a pointer to player that is initialized as a nullptr when the Dungeon is constructed. This function “news” a player into existence and assigns the player to an open position on the board, checking that there is no other Actor object standing in the returned position.

**void addObject(int r, int c, int key);**

This function is called at the beginning of the game and with each new level. It takes in three ints, the row and column of the position of the added object and a key that determines the type of object that is added to the vector of GameObject\*s. Additionally, it updates the dungeon map with the newly added object.

Pseudocode:

*Based on the input “key,” add the object that corresponds to that value*

*key 1: add short sword*

*key 2: add long sword*

*key 3: add mace*

*key 4: add strength scroll*

*key 5: add dexterity scroll*

*key 6: add armor scroll*

*key 7: add health scroll*

*key 8: add teleport scroll*

*key 9: add magic fangs*

*key 10: add magic ax*

**void addMonster(Coord pos, int key);**

This function is called at the beginning of the game and with each new level. It takes in a Coord describing the position of the monster and a key that determines the type of monster that is added to the m\_Monster member vector of Actor\*s. This follows similarly to the addObject() function:

Pseudocode:

*Based on key, add corresponding monster*

*key 1: add goblin*

*key 2: add snakewoman*

*key 3: add bogeyman*

*key 4: add dragon*

**void removeDead();**

At the end of each turn, the dungeon needs to check the status of monsters, deleting monsters from the board to update the game status. This function iterates through the m\_Monster member vector of Actor\*s and checks the HP status of each monster by using the getHP() accessor function from the Actor class. If the HP is less than or equal to 0, the monster is removed and deleted from the vector. The Game class calls this function and to update the board for each turn.

Pseudocode:

*Iterate over all monsters*

*If monster HP is <= 0*

*Trigger drop based on monster type*

*Delete/remove monster*

*continue to next monster*

**bool checkDead();**

This function is essentially to checking the health status of the player in order to update the game status. The current HP of m\_Player is checked with the getHP() accessor function from the Actor class. If the HP is less than or equal to 0, the game ends. The Game class calls this function and changes the player movement key to ‘q’ signifying the end of the game. This is similar to the previous removeDead() function.

**void display(string msg);**

This function is called by the play() function to update the dungeon after each turn is taken. This function takes in a string value that is displayed after the updated dungeon is displayed. It also checks if the game has been won by checking the Boolean value win. Order is important to ensure that the objects are displayed with correct precedence

Pseudocode: *clear current display  
create grid to hold current dungeon screen*

*Iterate over objects and set items*

*Set the staircase or idol depending on level*

*Set player*

*Iterate over monsters and set monsters*

*Display current dungeon*

*Display player statistics*

**Player Mechanics**

**string moveAttackPlayer(char direction);**

This function determines if the player will move or attack a surrounding monster. As the play mechanism is the same for both an attack and a move, this function uses the specific input character to check if the space is open (in that case move), occupied by a monster (attack), or a wall (do nothing). This returns a string that contains the turn action to be displayed on the updated dungeon screen.

Psuedocode:

*make value to determine if attack or move*

*for each direction: check if new space is open*

*if monster present, attack = true*

*if not, move player to spot*

*if attacking*

*iterate over monster and identify the attacked monster*

*player attacks monster*

*return display string*

**char viewInventory();**This function displays the player’s current inventory. It then takes in a user-defined character to redisplay the dungeon, which completes the player’s turn.

Pseudocode:

*Clear screen to display inventory*

*Make iterator and go through player inventory*

*Use ASCII and typecasting to print inventory spot + item name*

*Take user-defined character*

*Return to screen*

**string readScroll(); AND string wieldWeapon();**

Both of these functions are similar in that they require the player to view their inventory and select an object to be wielded or read. By building off of the viewInventory() function, this receives the player input character that defines the scroll or weapon that is wielded. This function returns a display string describing character action.

Psuedocode for readScroll():

*Call viewInventory() and return char as inventory spot*

*If invalid char (out of bounds/not valid), return empty string*

*Identify inventory item*

*If a scroll,*

*read scroll and perform action updating player stats*

*delete and remove scroll from inventory*

*add to display string*

*If a weapon, add message to string (cannot read weapon)*

*Return string*

Pseudocode for wieldWeapon();

*Call viewInventory() and return char as inventory spot*

*If invalid char (out of bounds/not valid), return empty string*

*Identify inventory item*

*If a scroll, add message to string (cannot wield scroll)*

*If a weapon,*

*update player’s weapon*

*add to display string*

*Return string*

**string pickUp();**

This function allows the player to pick up an object that has spawned on the floor of the dungeon or has been dropped by a monster. This function is called by Game when the player inputs a ‘g’ character. It returns a display string describing the turn action.

Pseudocode:

*create empty display string*

*if on idol*

*update display string*

*update win status to true*

*return string*

*if on item*

*if inventory full*

*update display string (inventory full, cannot pick up)*

*otherwise, identify object from vector of floorItems*

*add object to player inventory (player-defined function)*

*erase pointer to object from floorItems vector*

*update display string*

*return display string*

void cheat();

This function allows for the player to essentially “cheat,” and was utilized heavily in testing. This function is fairly simply and just asks the player to set m\_HP and maxHP to 50 and to set m\_strengthPts to 9.

**Monster Mechanics**

**string playMonsters();**

This function iterates through the vector of monsters and in turn, makes each take its turn (attack, move, remain still). This function is called by Game after playing the player’s turn and returns a display string containing the monster’s turn actions

Psuedocode:

*Make empty display string*

*For each monster:*

*for dragon, update HP based on 1/10 probability*

*if surrounding space (range = 1) has player*

*if player identified, attack*

*otherwise,*

*if snakewoman in range of player (3)*

*move towards player*

*if bogeyman in range of player (5)*

*move towards player*

*if goblin in range of player (as defined in call)*

*clear previous path from queue*

*find ideal path with recursive search*

*move towards player if a path is found*

**void moveMonster(Actor\* monster);**

This function is called by playMonsters() when a monster should move in response to player. This is the generic move function for the “dumb” monsters in the game. Goblins do not use this function. The algorithm for this function is described in part 4 of this report.

Pseudocode:

*get player position*

*get monster position*

*comparing player position to monster position (described in part 4)*

*move based on relative location*

**int goblinSearch(char map[18][70], Goblin\* monster, int r, int c, queue<Coord> path, int shortest);**

This function is the goblin search recursion function. It is detailed more thoroughly in part 4 of this report. The goblin search function is utilized by the playMonsters() function to move the goblin to the appropriately found spot. Pseudocode can also be found in part 4.

**Helper Functions**

**void setCellStatus(int r, int c, char status);**

**char getCellStatus(int r, int c);**

These two functions are essentially accessors and mutators for the maze, allowing for the gameplay to update the dungeon more effectively.

**Coord openSpot();**

This function returns a random open spot on the board is intended to help with placing game objects, the staircase, and the golden idol.

Pseudocode:

*Randomly select a row and column within dungeon dimensions*

*Check if the spot is actually empty*

*Return the Coord*

**bool inRange(int range, Coord\* playerPos, Coord\* monsterPos);**

This function is intended to make implementing monster movement much easier. As all monsters only move when the character is within a certain range, this returns true if the player is within range. It is used in playMonsters()when determining if a monster should move or remain still.

Pseudocode:

*Calculate difference between player and monster (steps)*

*sum absolute value of difference in rows and columns between player and monster*

*If difference <= range, return true*

*return false otherwise*

**bool noActor(int r, int c);**

This function is intended to make implementing monster movement much easier. As all actors cannot walk through other actors, this function checks if an actor is present at a specific spot. it is used in playMonsters() and playAttackPlayer().

Pseudocode:

*If there is a player at input coordinates, return false*

*For every monster*

*if monster is at input coordinates, return false*

*return true*

***Game Class***

The Game class was created with the purpose of controlling the flow of the gameboard. Game has one private member variable, a pointer to Dungeon which contains information regarding Mini-Rogue’s characters, objects, and movement.

Game implements the following member functions in addition to a constructor and destructor:

**void play()**

play() is responsible for adding a player to the board, which signifies the beginning of the game. This function takes in a character from the player each turn, calls the takeTurn(char move) function to play one round of the game. play() also checks for if the player dies before winning and updates the board display by calling display(string msg) function for the dungeon. This function runs until the game ends.

Pseudocode:

*Add player to game*

*Display initial game board for player  
while player input is not to quit*

*get player input*

*play one round of game based on player input*

*check if player is dead*

*quit game*

*update dungeon display based on display string*

**string takeTurn(char move)**

takeTurn(char move) reads in the player’s command (represented by a character) and calls all appropriate gameplay functions based on this action from Dungeon in the order described by the game rules. takeTurn() is not responsible for moving individual game components; Dungeon is. However, this function helps streamline play() returns a string containing game updates for the display function.

Pseudocode:

*Get player for ease of function*

*Make empty string*

*If player is not asleep*

*move player based on input (special input and movement)*

*update display string*

*if asleep*

*reduce sleep timer by one*

*remove dead monsters from board*

*play monsters and update display string*

*return display string*

1. **Algorithms**

***Making Rooms and Corridors***

While the algorithm to make a room is fairly simple, the corridors were more complicated. The following algorithm was utilized.

Pseudocode for makeRoom():

*make value to determine if room is made*

*randomly select start point of room*

*randomly select dimensions*

*check that room falls within grid (start + dimensions < grid dimensions for r and c)*

*if not, regenerate start point*

*check that room does not overlap with another room*

*iterate over whole proposed room*

*if any of current dungeon is open, don’t make room*

*if room can be made*

*add room*

*return if room was made*

Pseudocode for makeMaze():

*Clear last level  
Randomly generate number of rooms for level  
Add rooms using the makeRooms() function  
Add corridors:*

*Iterating over the rooms vector*

*Retrieve midpoints for current and next room*

*if room 1 is above room 2*

*and to the left*

*make corridor digging down and to left to connect midpoints*

*and to the right*

*make corridor digging down and to right to connect midpoints*

*and is directly above*

*make corridor digging down to connect midpoints*

*or room 1 is below room 2*

*and to the right*

*make corridor digging left and up to connect midpoints*

*and to left*

*make corridor digging right and up to connect midpoints*

*and directly below*

*make corridor digging up to connect midpoints*

*or rooms are horizontally lined up*

*and room 1 is to right of room 2*

*make corridor digging from midpoint of room 1 to room 2*

*and room 1 is to left of rom 2*

*make corridor digging from midpoint of room 2 to room1*

***Moving Monsters (Snakewomen and Bogeymen)***

**void moveMonster(Actor\* monster);**

This describes the specific algorithm used to identify the direction the “dumb” monsters should move to advance towards player.

Pseudocode:

*get player position*

*get monster position*

*comparing player position to monster position:*

*if player below monster*

*monster moves down*

*if player to right of monster*

*monster moves right*

*if player above monster*

*monster moves up*

*if player to left of monster*

*monster moves left*

***Recursive Goblin Search Function***

The recursive goblin search function is responsible for identifying if a path to the player is possible within a given range. The search function takes in the dungeon map (to be updated with points), the start position of the current search, the queue represented the path, and an int representing the shortest variable

Pseudocode:

*Mark current position on dungeon map*

*Get player position*

*If position of goblin is same as player*

*If the current path size is shortest one found*

*Update queue representing path in Dungeon (stored outside of function)*

*Return new shortest path size*

*Otherwise*

*Mark current position back to unvisited*

*Return the previously found shortest path size*

*If current path is longer than shortest path found*

*Return previously found shortest path size*

*For each direction (up, down, left, right):*

*If the next spot to be visited is open/unvisited*

*Recursively call function with this new spot*

*Mark position back to unvisited*

*Return shortest path length*

1. Known bugs and inefficiencies

Based on the compiler used, the recursive goblin search function may inefficient at smell ranges of 15 hindering gameplay. It definitely works, it is just slow.