**CS 32 - Project 3 Report & Design Documentation**

Non-Trivial Algorithms

1) Goblin Recursive Function

* **The goblin recursive function is named “findPath()” and is located in “Level.cpp”**. The function will return the optimal path length from the goblin to the player as long as the path length is less than the goblin smell distance. If the optimal path length from the goblin to the player is more than the goblin smell distance, the function returns 10,000 instead. The recursive goblin function “findPath()” is called 4 times in goblin’s “takeTurn()” function and returns the optimal path length if the goblin traveled up, down, left, and right. The shortest path length is then chosen from the 4 path lengths in the goblin’s takeTurn() function and the goblin moves in that direction. However, if all 4 path lengths returned by the findPath() function are greater than the goblin’s smell distance (ie findPath() returned 10,000) then the goblin does not move. The goblin’s “takeTurn()” function is located in “Actor.cpp”.
* The function takes in a copy of the current level layout, the position had the goblin moved in a certain direction (eg if the goblin moved left, the function takes in the row and column of the position after the goblin had moved one to the left), the position of the player, an int called pathLength that is incremented each time the function is recursively called, and a char signifying the direction that the goblin moved.
* Goblin Recursive Function pseudo code:
  + 1st base case - if the start and end position are equal, you’re done and return path length  
    2nd base case - if the goblin starts out by trying to move left, up, right, or down and immediately moves into a non-open position, then the goblin cannot move in that direction and the optimal path does not start in that direction  
    3rd base case - the current path length is greater than goblin smell distance, then return 10000 (ie the goblin is too far to “smell” the player0)  
    4th base case - if the shortest possible path between the goblin and player is more than the goblin smell distance, then goblin cannot smell player and return 10000  
    create 4 ints tracking the path length, one for each direction the goblin can travel  
    if the goblin moved down  
     recursively call findPath only for the goblin moving down, left, and right.  
     Set 3 of the 4 ints created to what findpath returns for each direction.  
     Moving up right after moving down is backtracking and not optimal  
    if the goblin moved up  
     recursively call findPath only for the goblin moving up, left, and right.  
     Set 3 of the 4 ints created to what findpath returns for each direction.  
     Moving down right after moving up is backtracking and not optimal  
    if the goblin moved left  
     recursively call findPath only for the goblin moving up, down, and left.  
     Set 3 of the 4 ints created to what findpath returns for each direction.  
     Moving right immediately after moving left is backtracking and not optimal  
    If the goblin moved right  
     recursively call findPath only for the goblin moving right, up, and down.  
     Set 3 of the 4 ints created to what findpath returns for each direction.  
     moving left immediately after moving right is backtracking and not optimal  
    compare all 4 path lengths and return the smallest

2) Function to randomly make rooms

* The algorithm is called “makeRooms()” and is located in “Level.cpp”
* Pseudocode:
  + choose a random int signifying how many rooms to make  
    The randomly made number of rooms signifies which if statement to go into  
    randomly choose the starting row and column for each room  
    for each room, based on the room’s starting row and column, make the 2D character array m\_level[][] populated with empty space  
    choose two rooms  
    randomly choose a row or column number shared by the two rooms  
    create a corridor between the two rooms, using this random row/column  
    repeat until all rooms have corridors between them

3) A function called “Play()” located in “Game.cpp” pseudocode:

create a string to that prints out player and monster messages

a bool that tracks if the player does an action that needs to be printed

a bool that tracks if monsters did an action that need to be printed

do

if the bool is to print a message

print the message

reset the bool to false

get a character from the user

if the player is not asleep

if the char is q

quit game

if the char is h, j, k, or l

call player’s take turn function (to move or attack)

if the char is g

pick up object function

if the car is i

call player’s display inventory function

if the char is w

call player’s wield weapon function

if the char is r

call the player’s read scroll function

if the input is c

call the player’s cheat function

if the player is asleep

decrement player’s sleep time

clear the board of dead monsters by calling a function

move all alive monsters by calling a function

heal the player by calling a function

while the player is not dead

// outside of the do while, the player is dead

display the level one last time

print out the last message

while the inputted char is not q

don’t do anything

// program ends after user inputs q

4) Dungeon constructor located in “Dungeon.cpp” pseudocode and documentation:

point to the game

set current level as 0

call level’s constructor and return a pointer to the new level

5) Dungeon’s function to make a new level called “newLevel()” located in “Dungeon.cpp” only called after going up a staircase

increment current level int

delete the previous level

create a new level and return a pointer to it

find a random open position in the new level to place the player

6) Level constructor located in “Level.cpp” pseudocode and documentation:

* makes the rooms first. Then randomly adds the appropriate number of weapons/scrolls and monsters, and staircase/idol. Then gets random coordinates for player.

make entire 2D char array representing level walls

call makeRooms() function to create rooms by filling 2D char array with white space

randomly choose number of objects to add to level

assign random int for weapon or scroll

for each weapon

if int designates weapon

add weapon

if int designates scroll

add scroll

randomly choose number of monsters to add

for each monster

if level 0 or 1

add snakewomen or goblin

if level 2

add snakewomen, goblin, or bogeymen

if level 3 or 4

add snakewomen, goblin, bogeymen, or dragon

choose random location for idol or staircase

if level 0 - 3

add staircase

else if level 4

add idol

choose random position for player

7) Level Destructor located in “Level.cpp” psuedocode and documentation:

* deletes the staircase/idol. deletes all weapons/scrolls not picked p and deletes all monsters not killed.

delete staircase or idol

for each object on level

delete object

for each monster on level

delete monster

8) display function located in “Level.cpp” pseudocode and documentation:

* updates 2D char array representing level with new position of objects picked up or dropped, newest position of player after moving, and newest position of monsters after moving. Then uses a for loop to print out the 2D char array

9) validMove() function located in Level.cpp → determines if an actor can move in that direction by checking if there is a player, monster, or wall in the direction the actor wants to move. If the actor can move in that direction, the function returns true

10) createPlayer() function located in Level.cpp → gets a random open position for the player on the level then uses the keyword new to create a player at that position. Only called once since there is only 1 player per game

11) pickUpObject() function located in Level.cpp → picks up the idol or a weapon/scroll. if the idol is picked up, the appropriate message prints and the game waits for the user to enter q before exiting. If a weapon or scroll is picked up the appropriate message is added to the string printed at the end of each turn. If the knapsack is not full, the item is picked up and pushed into the player’s inventory vector of game objects and erased from the vector of game objects displayed on the level. Otherwise, the item is not picked up

12) addInteractableObject() function located in Level.cpp → called when a new level is constructed. Dynamically creates a new weapon or scroll with random coordinates to be placed on the level. The pointer to the new game object is pushed to the level’s vector of game object pointers.

13) addMonster() function called in Level.cpp → finds a random free position to place the monster. Dynamically instantiates a new monster at that position and pushes a pointer to the monster into the level’s vector of monster pointers.

14) clearDeadMonsers() located in Level.cpp → deletes every monster with hit points less than or equal to 0 after the player’s turn

15) monsterDropItem() located in Level.cpp → called in addMonster() function before the dead monster is deleted. Depending on the monster that was killed (ie the monster that is going to be deleted using the keyword *delete*) a new weapon or scroll is dropped at the position of the monster if there is not already an object there. A pointer to the newly dropped object is pushed to level’s vector of game object pointers

16) isObjectAtSpot() function in Level.cpp → called in the monsterDropItem() function before a monster drops an item. Returns true if an object is at that position signifying a monster cannot drop an item at that position. Returns false if an object is not at that position, signifying a monster can drop an item at that position after dying.

17) moveMonster() function in Level.cpp → for each “alive” monster (ie each monster with hit points greater than 0 and have not been deleted yet) in level’s vector of monster pointers, calls the monster’s takeTurn() function. takeTurn() decides if the monster should attack, move, or do nothing.

18) move() function located in Actor.cpp → has a char parameter called direction signifying how to modify the actor’s current row and column position. The actor’s row and column position is modified if the actor can move in the direction signified by the char parameter. if not, the actor’s position is not modified. move() called validMove() to determine if moving in that direction is possible and if it should modify the actor’s position accordingly.

19) bool function called isMonsterAtPosition() located in Actor.cpp → called in player’s takeTurn() function. If a monster is at a certain position, the function returns true. This function lets player’s takeTurn() function know if the player should attack the monster at that position or move to that position.

20) attack() function located in Actor.cpp documentation and pseudocode→ determines if the attacker’s weapon is magic fangs of sleep or not. Then calculates attacker and defender points. If the attacker points are enough, the defender’s points are decremented accordingly and the string to be printed out after each turn is modified with the appropriate message. The bool signifying that a message should be printed out is set to true. If the attacker’s weapon is magic fangs of sleep, the the defender is put to sleep with a 0.20 chance. Pseudocode below:

dynamic cast the attacker’s weapon pointer to magic fangs of sleep

calculate defender and attacker points

if the magic fangs of sleep pointer is not null pointer (then weapon is fangs of sleep)

if attacker points are greater than defender points (according to formula in spec)

decrement defender hit points

modify message to print out

bool that says a message should be printed out is set to true

bool with 0.20 chance of being true

if that bool is true

increment defender’s sleep points according to formula in spec

if magic fangs of sleep pointer is a null pointer

if attacker points are greater than defender pointers

decrement defender hit points

modify messages to print out

bool that says a message should be printed out is set to true

if attacker points is less than defender points

no attack occurs (defener’s hit points are not decremented)

modify message to print

set bool that says a message should be printed out to true

21) heal() function in Actor.cpp → heals the dragon and player each turn with a chance of 0.10. The 0.10 random chance is determined by a provided function in utilities.cpp called trueWithProbability()

22) Goblin’s takeTurn() function located in Actor.cpp → The function determines if the goblin should attack, move, or do nothing each turn. If goblin is asleep, decrement sleeptime and return. If the goblin is right next to the player, the goblin will attack the player then the function returns. Otherwise, the recursive goblin function named findPath() is called 4 times, returning 4 different ints. Each of the 4 ints represents the optimal path length from the goblin to the player had the goblin moved left, right, up, and down. If the goblin is too far, the findPath() function instead returns 10000. The smallest of the 4 ints is chosen and the goblin moves in the corresponding direction. If all ints returned by the recursive findPath() is 10000, then the goblin does not move and the function returns without having the goblin do anything.

23) Snakewomen’s takeTurn() function located in Actor.cpp → The function determines if the snakewomen should attack, move or do nothing each turn. If the snakewomen, is alseep then decrement the snakewomen’s sleeptime and return. If the snakewomen is next to the player, then the snakewomen attacks the player by calling the attack() function (also located in Actor.cpp). Since the snakewomen can “smell” through walls, determine the distance between the player and snakewomen by determining the row and column distance between the snakewomen and player. If the distance is less than three, the snakewomen moves closer to the player. If not the function returns without having the snakewomen do anything.

24) Bogeymen’s takeTurn() function located in Actor.cpp → The function determines if the bogeymen should attack, move, or do nothing. If asleep, then decrement sleep time and return. If next to the player, then attack the player by calling the attack() function and return. Since the bogeymen can “smell” through walls, determine the distance between the player and bogeymen by seeing the difference in row and column distance. If it’s less than 5, move towards the player and if not the function returns without the bogeymen doing anything.

25) Dragon’s takeTurn() function located in Actor.cpp → Determines if the dragon should attack or not. Never moves the dragon because per the spec the dragon never moves. call the heal() function to heal the dragon since the dragon is healed each turn. If the dragon is asleep, then decrement its sleep time and return. If the player is next to the dragon, the dragon attacks the player and the function returns. Otherwise, the function returns without having the dragon do anything during its turn.

26) Player’s destructor located in Actor.cpp → uses a for loop to loop through the actor’s vector of weapons and scrolls and calls delete on each item in the vector.

27) Player’s displayInventory() function located in Actor.cpp → calls clearScreen() function. Uses a for loop to write out the name of each weapon or scroll in the player’s vector of game objects

28) player’s wieldWeapon() function located in Actor.cpp → calls clearScreen() function. Then prints out player’s inventory. Calls getCharacter() function (provided by utilities.cpp). If the character inputted corresponds to a weapon in the inventory, the player’s pointer that points to the current weapon its wielding is changed. Otherwise, the function returns without changing the player’s weapon (ie without changing the pointer that points to the current weapon the player wields). The messageToPrint is updated and the bool representing that a message needs to be printed is set to true.

29) player’s readScroll() function located in Actor.cpp → calls clearScreen() function and prints out the inventory. Calls getCharacter() function and if the inputted character corresponds to a scroll in the inventory, the player’s stats are incremented accordingly. Delete is then used to deallocated the memory used by the scroll that was just read. The messageToPrint is updated and the bool representing that a message needs to be printed is set to true.

30) player’s takeTurn() function located in Actor.cpp → Determines if the player should attack or move in that position. If a monster is at the position the player is trying to move, the player attacks the monster. Otherwise, the player will move to that position.

Purpose and Major Responsibilities of Each Class and Relation to Each Class

1) Overall design of the program

* I have 5 base classes and their derived classes (listed below)
  + a) Game class
  + b) Dungeon class
  + c) Level Class
  + d) Actor Class → Actor is the base class. Player is a derived class of actor. Monster is a derived class of Actor. Snakewomen, Bogeymen, Goblin, and Dragon are derived classes of Monster.
  + e) Interactable Object Class → Interactable Object is the base class. Progression objects are a derived class of interactable objects. Staircases and golden idols are a derived class of progression objects. Weapons are a derived class of interactable objects. Shortswords, maces, magic axes, long swords, and magic fangs of sleep are a derived class of the weapons class. Scrolls are a derived class of the interactable object class. Scrolls of health, dexterity, armor, strength, and teleportation are derived classes of the scroll class.
* The major responsibility of each class and their relation to each other are explained below

2) Game Class

* The game class creates the dungeon and the player and has a pointer to the dungeon and player.
* The game class constructor will create the dungeon. When dungeon is created, the dungeon constructor will call the level constructor. After the level constructor is called by the dungeon constructor, the level constructor will randomly create rooms and corridors, weapons, scrolls, a staircase or idol, and monsters and place them into the level. The level will also choose random coordinates that will become that player’s initial position on the level. Note that the level is constructed before the player since we cannot assign the player a valid position until we know the free and non-free positions in the level
* The game class acts as the “driver” for the game. The game class has a do while loop in its play() function that loops until the player is dead. The do while loop will call display() to display the current level, call getCharacter() to get a command from the user and then call the appropriate function to wield weapons, move the player, attack monsters, etc. The do while loop will also appropriately print messages such as “Player attacks the snake woman with short sword and misses”. Outside of the do while loop, the player is dead (ie the player has hit points less than or equal to 0). Outside of the do while, the game class’s play() function will print out the correct message and wait until the user hits q to quit the game.

3) Dungeon class

* The dungeon class keeps track of the current level the player is on. The dungeon class also instantiates a new instance of the level each time the game is created and each time the player progresses to a new level by going up the staircase. The dungeon class also deallocates memory used by a level that the player is no longer on.

4) Level class

* The level class is responsible for creating a 2D char array representing the level that the user sees. A new instantiation of level is done when the level class constructor is called in the dungeon class’s constructor and when dungeon’s newLevel() function is called. Every time the player goes up the staircase the dungeon’s newLevel() function is called which deallocates the old level and instantiates a new level object.
* The level class is responsible for randomly placing all the monsters, weapons/scrolls, and staircase or golden idol onto each newly created level. It has a pointer or vector of pointers to all of these objects. The level class has function that deletes monsters that have been killed by the player, a function that adds a weapon or scroll that has been picked up by the player to the player’s inventory and deletes it from the level’s vector of object pointers
* The level class is responsible for deallocating the memory of each monster not killed by the player and each weapon or scroll not picked up by the player after the player moves to the next level and after the game is played.
* The level is also responsible for moving all alive monsters each turn so it has a function for this. Accordingly, the recursive function that finds the optimal path length from the goblin to the player is a member function of the level class. This function is called by the goblin’s takeTurn() function (as explained above).

5) Actor class

* The player and all monsters are derived classes of the actor class.
* All actors will have data members that signify their location, their stats, and a pointer to the weapon they are holding.
* The actor class has an abstract function called takeTurn() which decides what each actor will do each turn. Since each actor does something different each turn, the takeTurn() function is an abstract virtual function.
* The player has an additional vector data member that has pointers to all the objects in the player’s inventory. After the game is lost or won, the player’s destructor will delete all dynamically allocated weapons and scrolls that are in the player’s inventory.
* The Monster class is a derived class of actor. Each specific monster (eg. dragon, bogeymen, snakewomen, and goblin) are derived classes of the monster class. Each specific monster has their own takeTurn() function since each monster does something different each turn.
* After a monster dies and drops a weapon or scroll, a pointer that points to that dynamically allocated weapon or scroll is added to the level’s vector of weapon and scroll pointers.
* The level class also has a vector of monster pointers that point to each monster on the level. Each monster is deallocated and each monster pointer is erased from the vector after a player kills the monster. All the monsters are deallocated when the player goes up a staircase and a new level is constructed.

5) Interactable Object class

* The interactable object class encompasses the weapons, scrolls, staircase, and golden idol.
* Each interactable object has a char denoting its symbol (the ? ) > or & char). Weapons will have their own data members signifying its action, the weapons name, its dexterity bonus, and its damage amount. Scrolls have their own data members signifying the scroll name, the scrolls action, a char signifying the scroll type, and an int called m\_enhnace that signifies how much to increase the player’s stats by. For the scroll of teleportation, m\_enhance is irrelevant and instead the teleportation scroll uses the freePosition() function to find a free random position to teleport the player.
* Weapon is a derived class of the interactable object class. Each specific weapon (eg. mace, long sword, etc.) is a derived class of the weapon class. Each actor has a pointer to the weapon they are currently wielding. The pointer that points to the weapons the actor is currently wielding only changes for the player, since the player can wield different weapons but each monster wields the same weapon for the duration of the game. In addition to a pointer pointing to the current weapon the player is wielding, the player has a vector of intractable object pointers that point to each weapon in the player’s inventory.
* Scroll is a derived class of the interactable object class. Each specific scroll (eg. teleportation, armor, etc.) is a derived class of the scroll class. The player’s inventory of interactable object pointers can point the scrolls that have been picked up by the player.
* The level class also has a vector of interactable objects pointers that point to a scroll or weapon that has been randomly placed on the level. After an object has been picked up, the player’s inventory now points to the scroll or weapon and the object is erased from level’s vector of interactable object pointers. Also, when a level is destructed all the dynamically allocated weapons and scrolls not picked up by the player are deleted by the level’s destructor.

Known bugs and serious inefficiencies

1) The recursive goblin function is a little slow but not uncomfortably slow. If the level has been generated with goblins on it, you have to wait about 1-3 seconds after inputting your command.