The goblin movement function is called

char Goblin::smell(Player\* p, char maze[18][70])

It relies on recursive behavior, found in

bool Goblin::pathExists(char path\_maze[18][70], int er, int ec, queue<Coord> coordQueue, int max\_step)

Both functions are found in Actors.cpp

The design for my program is as follows:

Most game logic (User input, monster interaction with rooms, map generation, cross entity interaction) is handled by the Levels class, with some functionality (such as inventory management and item dropping) being handled by other more applicable classes in conjunction with helper functions within the Levels class. Each Level has a vector of objects and monsters, as well as a pointer to a player, and an exit object. Each game has 5 level objects, and a player. The game passes the player through each level object, transporting the player to the next level when it is completed, or ending the game when the player dies/quits.

The functionality of each of class is as follows.

The purpose of each class is as follows:

IN ACTORS.H

Actor

Actor provides base functionality for all actors (monsters and players), in the form of basic functionality such as movement, manipulating and returning statistics (such as health and armor), and storing/returning the current wielded weapon.

Actor is the base class for Monster and Player, and is thus called upon by every function which needs to access or manipulate any stat applicable to all actors (position, weapon, health, max health, armor, strength, dexterity, sleep time, death status, map symbol, name). Aside from these getter and setter functions, Actor only has one non-trivial function:

string attack(Actor\* target);

This function decides whether the calling actor hits the target actor, and if so, calculates the damage inflicted. In the special case where the actor’s current weapon is magic fangs, this function decides whether the target is put to sleep, and if so, for how long. The function returns a corresponding string for the action which took place.

Player

Player is a subclass of Actor, and provides additional functionality to the Actor class specific to the player. The Player class adds to the Actor class an object inventory in the form of an object vector, and a “used teleport” Boolean, used to indicate whether the player has used a scroll of teleportation. Aside from trivial getter and setter functions, this class adds:

string wield();

This function displays the inventory, and upon user input, attempts to equip the selected item in the player’s weapon slot. The function returns a corresponding string for the action which took place.

string pickup(Object\* item);

This function attempts to add the parameter object to the player’s inventory, and returns a corresponding string for the action which took place. This function is called upon by the level class whenever a player attempts to pickup an item.

string read();

This function displays the inventory, and upon user input, attempts to use the selected item with the player’s useScroll function. The function returns a corresponding string for the action which took place.

void useScroll(Scroll\* s);

This function identifies the parameter scroll’s ID using the scroll’s ID getter function, and applies the corresponding to the player, deleting the scroll after successful use.

void viewInventory();

This function clears the screen and displays a list of the items in the user’s inventory, labeled from a to z.

string input\_handler(char input);

This function is called upon by the level class for inventory specific inputs (ie. W, r, and i). The function then calls upon the player’s wield, read, and inventory functions, depending on the input, and returns the result of the function call (returns empty string if the input was not w, r, or i).

Monster

Monster is a subclass of Actor, and provides additional functionality to the Actor class specific to monsters. The Monster class adds to the Actor class an additional member variable smell distance, which can be called on through its respective getter function. Aside from trivial getter and setter functions, this class adds:

Object\* onDeathDrop()

This function is a pure virtual function, which is implemented by each subclass of Monster differently

char smell(char grid[18][70], Player\* p);

This function searches the parameter grid for a player within the monster’s smell radius, by tracing diagonal lines northeast, southeast, southwest, and northwest of the monster’s current position, up to the monster’s maximum smell radius (detailed more below). This function returns a direction character (h,j,k,l) if the player is detected and the monster can move closer, or ‘x’ to indicate no move. This function is called up on by the levels monster\_move function

Bogeyman

Bogeyman is a subclass of Monster. It allows for the standardized generation of bogeymen with the starting stats defined by the game specification, and implements the pure virtual death drop function defined in the Monster class

Object\* onDeathDrop();

This function rolls to determine whether to return a nullptr (no item drop), or a newly generated magic axe, given the Bogeyman’s current position and the position of items in the level. This function is only called by the level class.

Snakewoman

Snakewoman is a subclass of Monster. It allows for the standardized generation of snakewomen with the starting stats defined by the game specification, and implements the pure virtual death drop function defined in the Monster class. This function is functionally similar to the one defined in Bogeyman (although they are completely separate and unrelated), with the only difference being the weapon returned (magic fangs).

Dragon

Dragon is a subclass of Monster. It allows for the standardized generation of dragons with the starting stats defined by the game specification, and implements the pure virtual death drop function defined in the Monster class.

Object\* onDeathDrop();

This function rolls to determine which of the 5 scroll objects (defined later) the function will return upon execution, given the Dragon’s current position and the position of items in the level (None if there is already an item in the spot). This function is only called by the class level.

Goblin

Goblin is a subclass of Monster. It allows for the standardized generation of goblins with the starting stats defined by the game specification, and implements the pure virtual death drop function defined in the Monster class. This function is functionally similar to the one defined in Bogeyman (although they are completely separate and unrelated), with the only difference being the weapon returned (short sword). Additionally, the Goblin class implements its own version of the smell algorithm, in two separate parts

bool Goblin::pathExists(char path\_maze[18][70], int er, int ec, queue<Coord> coordQueue, int max\_step)

This function identifies the fastest path to the player using breadth first search recursive behavior, if such a path exists

char Goblin::smell(Player\* p, char maze[18][70])

This function translates the queue generated by pathExists into a direction character, for use with Level’s move\_monster function

IN OBJECTS.H

Object

Object provides base functionality for all objects (scrolls, exits, weapons), in the form of basic functionality through getters and setters which return and manipulate various member variables (“on ground” status, position, action, name, symbol).

Object is the base class for Weapon and Scroll.

Weapon

Weapon adds additional functionality for all weapons (eg. Mace, long sword) by adding two additional member variables, damage and dexterity, publicly callable through Weapon’s getter functions. Besides additional getter and setter functions, the Weapon class itself does not add any additional functionality.

Weapon is a subclass of Object and a base class for Mace, S\_Sword, L\_Sword, Axe, and Fang.

Mace

Mace is a subclass of Weapon, and serves as a standardized object type from which maces can be constructed with the stats defined by the game specification

S\_Sword

S\_Sword is a subclass of Weapon, and serves as a standardized object type from which short swords can be constructed with the stats defined by the game specification

L\_Sword

L\_Sword is a subclass of Weapon, and serves as a standardized object type from which long swords can be constructed with the stats defined by the game specification

Axe

Axe is a subclass of Weapon, and serves as a standardized object type from which magic axes can be constructed with the stats defined by the game specification

Fang

Fang is a subclass of Weapon, and serves as a standardized object type from which magic fangs can be constructed with the stats defined by the game specification

Scroll

Scroll adds additional functionality for all scrolls by adding an additional member variable, type. Besides additional getter and setter functions, the Scroll class itself does not add any additional functionality.

Scroll is a subclass of Object and a base class for Teleport, Armor, Strength, Health, and Dexterity.

Teleport

Teleport is a subclass of Scroll, and serves as a standardized object type from which scrolls of teleportation can be constructed with the corresponding identifier (type).

Armor

Armor is a subclass of Scroll, and serves as a standardized object type from which scrolls of enhance armor can be constructed with the corresponding identifier (type).

Strength

Strength is a subclass of Scroll, and serves as a standardized object type from which scrolls of raise strength can be constructed with the corresponding identifier (type).

Health

Health is a subclass of Scroll, and serves as a standardized object type from which scrolls of enhance health can be constructed with the corresponding identifier (type).

Dexterity

Dexterity is a subclass of Scroll, and serves as a standardized object type from which scrolls of enhance dexterity can be constructed with the corresponding identifier (type).

IN GAME.H

Game

Game provides base functionality for the playthrough of the entire game. The Game class contains a private member variable for keeping track of the current level, a private member variable for storing the passed goblinSmellDistance, a Player pointer, and a vector of Level pointers. Its only member function (besides constructors and destructors) is the play function.

void play();

Executes a play session of the game, calling on the required public level functions, and looping until level 4 has been complete. Note that for optimization purposes, a new level is generated and added to the level vector at the end of each level until level 4, rather than each level being pre-generated at the start of the game.

This class relies on all other classes for functionality.

IN UTILITIES\_S.H

Coord

Coord provides functionality for a coordinate datatype, consisting of a row, column, and step member variable, for use with the Goblin pathExists function and the Level fill\_grid function. Aside from trivial constructors, destructors, and getter functions, this class does not add any functionality by itself.

room

room provides functionality for a room datatype, consisting of left, right, top, and bottom integer member variables, used for tracking the edges of a generated room. Its constructor is special in that it randomly generates each of these values, with a higher probability that a generated room will be wider than it is tall. Aside from trivial getter functions, the room class does not add any functionality on its own.

IN LEVEL.H

Level

Level provides most of the main game logic functionality, and as such features the most of the complex algorithms found in the game. The Level class consists of 2d character array, pointers to a player and an exit object, vectors of pointers to monsters and items, and member variables to track the current level, whether the level has been completed, and the number of items and monsters. Level’s functionality relies upon the functionality of Actor and its subclasses, Object and its base classes, room, and Coord.

NOTE: THESE ARE AN OVERVIEW OF THE PUBLIC FUNCTIONS OF LEVEL

Aside from trivial getter and setter functions, Level provides the following functionality:

Level(Player\* player, int goblinSmellDistance, int stage);

The Level’s constructor is responsible for the generation of the game level. It randomly generates a room, spawns items, spawns monsters, spawns the player, and spawns and exit. (NOTE THAT THESE DUTIES ARE FURTHER DIVIDED INTO SMALLER FUNCTIONS, DETAILED LATER)

void display();

This function loops through the character grid and prints each character, in order to display the level.

void update();

This function loops through its item and monster vectors, removing dead monsters or picked up items. The function then replaces every non-wall character (ie. Erasing the board), and then loops through its monster and item vectors, redrawing each item/monster, and redraws the exit and player. If the player’s teleport status is 1, the function respawns the player. Note that this function does not print anything to the console.

void player\_move();

This function takes the player’s keyboard input, and executes corresponding private/member variable functions in order to process player movement/action

void monster\_move();

This function loops through the monster vector, processing any movement made by monsters given the current state of the game.

NON-TRIVIAL ALGORITHMS

void Level::player\_move()

Checks if the player is asleep

if so, waits for an input before decrementing the player’s sleep timer.

rolls to decide whether the player will heal on this turn. Heal if so

store the player’s next input, and passes it to the player’s input handler.

If the input was an “I”

End the turn (return)

Otherwise, if the input handler returned a response

Add response to message buffer, end turn

If there was no response

If the input corresponds with a move (h,j,k,l)

Pass input to valid\_attack

Attack in that direction if valid

Otherwise pass input to valid\_move

Move in that direction if valid

End the trun

If the input was a g, the player’s pickup function is run and the turn ends.

If the input was a “>”, the player’s position is checked against the staircase position, and if they match, the level is marked as complete. Turn ends.

If the input was a “Q”, the level is marked as over, and the turn ends.

If the input was a “C”, the player’s cheat function is executed, and the turn ends.

If the input was anything else, the turn ends.

void Level::monster\_move()

If the level is complete, the turn ends.

For every monster in the level’s monster vector, the following operation is run

If the monster is asleep, decrement the sleep counter and end the turn.

If the monster can attack in any adjacent direction, attack in that direction.

If the monster is a goblin, check if the player is within rough range of the goblin. If so, attempt moving in the direction indicated by the goblin’s smell algorithm.

If the monster isn’t a goblin, attempt to move in the direction indicated by the monster’s smell algorithm

Update the board, and end the turn

bool Level::valid\_move(char dir, Actor\* a)

for any given passed character “dir”, check the following

If it is a valid movement character, check the space corresponding with a movement in that direction. If moving onto the space is possible, return true

Return false (occurs if the character is not a valid movement key or if the space is occupied)

bool Level::valid\_attack(char dir, Actor\* a)

If a is a player, use the player’s enemy set

Otherwise, use the monster’s enemy set

For any given passed character “dir”, check the following

If it is a valid movement character, check the space corresponding with a movement in that direction. If there is an actor from the current enemy set on that space, return true.

Return false (occurs if the character is not a valid movement key or if the space is not occupied by an enemy from the enemy set)

void Level::player\_attack(Player\* p, int x, int y)

Loop through the level’s monster vector and find the monster at x and y

Set this monster as the target

Attack the target

If the target is dead:

Check if the monster was standing on an item

If so, do nothing

If not, get the object returned by the target’s onDeathDrop function

If the object is not a nullptr, add the object to the level’s item vector

Object\* Level::getItem()

Set the return item as a nullptr

Loop through the item vector

If the item has the same position as the player, return the item

Return the return item (nullptr)

void Level::spawn\_monsters(int goblinSmellDistance, int stage)

Calculate the number of monsters to generate from the stage number

Run the following until the number of monsters = calculated number

Randomly find a valid spot (not occupied by other actors, not a wall)

If stage >= 3, generate from all of the following

If stage >= 2, generate from the first 3

Otherwise, generate from the first 2

Make a new goblin at the empty spot, passing goblinSmellDistance

Pass the goblin into the monster vector

Or make a new Snakewoman at the empty valid spot

Pass the snakewoman into the monster vector

Or make a new bogeyman at the empty valid spot

Pass the bogeyman into the monster vector

Or make a new dragon at the empty valid spot

Pass the dragon into the monster vector

void Level::spawn\_items()

Randomly decide the number of items to generate (2 or 3)

Run the following until the number of items = number

Randomly find a valid spot (Not occupied by other objects, not a wall)

Randomly generate one of 6 items at the spot

Short sword, long sword, mace, armor scroll, strength scroll, dexterity scroll, or health scroll

Add the item to the item vector

Randomly generate a valid spot (Not occupied by other objects, not a wall)

If the current level is 4, create an idol object at the spot, saving it in m\_exit

Otherwise, create a stair object at the spot, saving it in m\_exit

void Level::spawn\_player(Player\* p)

Find a valid spot (Not occupied by an actor, not a wall)

Set the player’s position to that spot

void Level::generate\_grid()

Make a grid full of walls

Randomly decide how many rooms to make

Make a room vector

Run until we have made enough rooms

Make a new room

Check that each space occupied by the new room is not already occupied by another room

If so, generate a new room

If no overlap was found, add the room to the room vector

Run for each room in the room vector

Check each other room to see if it is to the right of the current one

If it is, and there is a horizontal overlap, and it is less than 15 spaces away

From a random point within the horizontal overlap, create a corridor to the right until the other room is met

Check each other room to see if it is under the current one

If it is, and there is a vertical overlap, and it is less than 15 spaces away

From a random point within the vertical overlap, create a corridor downwards until the other room is met

void Level::test\_grid()

Run until exited forcibly

Generate a grid

Find an empty space in the grid

Fill the entire grid from that space, not passing through any walls

Check the entire grid for empty spaces

If there are empty spaces, remake the grid

If there aren’t any empty spaces, exit the loop

void Level::grid\_fill(int sr, int sc)

Pop the starting coordinate (at sr, sc) onto a queue

Mark the current position in the grid with an O

Run while the queue is not empty

Set the current position to the queue’s top coord

Pop the queue

If there is a valid space above

Add the space to the queue

Mark the space with an O

If there is a valid space above

Add the space to the left

Mark the space with an O

If there is a valid space to the right

Add the space to the queue

Mark the space with an O

If there is a valid space below

Add the space to the queue

Mark the space with an O

string Actor::attack(Actor\* target)

Generate the beginning of an attack string

Calculate the attackerPoints given the attacker

Calculate the defenderPoints given the target

Calculate the damage given the current weapon

If attackerPoints >= defenderPoints

Subtract the damage amount from the defender’s HP

Check if the current weapon is magic fangs

If the current weapon is not magic fangs

If the target is dead

Add to the attack string accordingly

Otherwise

Add to the attack string accordingly

If the current weapon is magic fangs

Roll between 1 and 5 to decide if the target sleeps

If the target is dead

Add to the attack string accordingly

Otherwise if rolled successfully

Set the target’s sleep timer to a random number (1,6)

Add to the attack string accordingly

Otherwise

Add to the attack string accordingly

Otherwise finish the attack string accordingly

Return the attack string

string Player::pickup(Object\* item)

If the passed item is a nullptr

Return an empty string

If the inventory size >= 25

Add the item to the inventory

Return the corresponding action string (dependent on item type)

Else

Return the inventory full message

void Player::useScroll(Scroll\* s)

Get the ID of the scroll

Change the corresponding player member variable values

Delete the scroll

string Player::wield()

View the inventory

Setup a return string

Get user input

If the ascii value of the input > 96

Subtract 97 from the input value and store as an int

If the converted number is less than the inventory size

Check if the item at the index is a weapon

If so, equip the weapon and add the corresponding message to return

If not, add corresponding message to return

Otherwise, do the same thing as above, but subtracting 65 instead of 97

Return the return string

string Player::read()

View the inventory

Setup a return string

Get user input

If the ascii value of the input > 96

Subtract 97 from the input value and store as an int

If the converted number is less than the inventory size

Check if the item at the index is a scroll

If so, use the scroll and add the corresponding message to return

If not, add corresponding message to return

Otherwise, do the same thing as above, but subtracting 65 instead of 97

Return the return string

string Player::input\_handler(char input)

If the input is I

View the inventory

Await keyboard input

Clear the screen

Return “viewed\_inv” Note: this is discarded the Level function that uses it

If the input is W

Return the message given by the wield function

If the input is r

Return the message given by the read function

Otherwise

Return an empty string

char Monster::smell(char grid[18][70], Player\* p)

Make a Boolean, set it to false

Run the following from x = 0 until smell distance, incrementing x after each run

Check the spaces x distance away to the northwest

If the space is the same space as the player, set the Boolean to 1

Check the spaces x distance away to the northeast

If the space is the same space as the player, set the Boolean to 1

Check the spaces x distance away to the southwest

If the space is the same space as the player, set the Boolean to 1

Check the spaces x distance away to the southeast

If the space is the same space as the player, set the Boolean to 1

Exit the loop if the Boolean is 1

If the Boolean is 1

If the moving to the east is possible and will bring the monster closer

Return l

If the moving to the west is possible and will bring the monster closer

Return h

If the moving to the south is possible and will bring the monster closer

Return j

If the moving to the north is possible and will bring the monster closer

Return k

Otherwise, return x

char Goblin::smell(Player\* p, char maze[18][70])

Run pathExists given the current scenario

If pathExists

If the moving to the east is possible and will bring the monster closer

Return l

If the moving to the west is possible and will bring the monster closer

Return h

If the moving to the south is possible and will bring the monster closer

Return j

If the moving to the north is possible and will bring the monster closer

Return k

Otherwise, return x

NOTE: I realize this part is not recursive. I could not figure out how to return the best path recursively (although my function IS able to find the path recursively). As a band aid fix to make the game seem more complete, I used the same movement algorithm found in Monster’s smell function. Given more time, I would certainly have loved to make this feature full

bool Goblin::pathExists(char path\_maze[18][70], int er, int ec, queue<Coord> coordQueue, int max\_step)

If the queue is empty

Return false

If the current position matches the player position

Return true

Pop the queue

For every direction

If the current step < max step

If a move is possible, mark the space, add the space to the queue

Return the result of the function rerun

void Game::play()

Run while the current level <= 4

Clear the screen

Display the current level

Run while the current level is not completed

Check if the player is dead, requesting Q if so

Run the current level’s player\_move function

End the game if the player exited

End the game if the player was on the 4th level and the level is marked as completed

Update the board

Run the current level’s monster\_move function

Clear the screen

Display the screen

Increment the game’s currentlevel variable

If the current level <= 4

Make a new level and add it to the game’s level vector

KNOWN BUGS/NON-IMPLEMENTED FEATURES/SERIOUS INEFFICIENCIES

* I ran into a case during a testplay where there were some empty spaces not connected to the rest of the maze. I didn’t notice them at first, so I’m unsure if it was a bug with level generation or something else. After numerous attempts, I could never recreate the problem
* Currently, the goblin movement algorithm does not return a usable list. As a result, it simply acts as a much more inefficient version of the regular smell algorithm, since it doesn’t return the best move. However, the Goblin can not smell past walls, but can smell around them.
* For whatever reason, the game runs fine in G32 – except no items or monsters are generated. The game works completely fine in Visual Studio.