Zena's CS231 Project 9- Hunt The Wumpus (Final Project)

Task

This project asked us to design a game based on the classic "Hunt The Wumpus" game. In this game, the player walks through rooms in search of the wumpus, and the rooms that are within two steps of the wumpus's lair are drawn red. This gives the player a hint in deciding which room to attack; if the player gets it right, the game is won, but if the wumpus was not hit, the wumpus hears the hunter and eats him. Otherwise, walking into the wumpus's lair directly will also cause the hunter to be eaten.

General Solution

This game required quite a few components, including a hunter, wumpus, and Vertex (a room) class, as well as logic to work with keystrokes. The game responds to certain keystrokes, upon which the components are updated, and the window refreshes its display. In my game, the arrow keys are used to move, and the q key can be used to exit. Also note that the spacebar can be used to equip and unequip the sword. While equipped, pressing in a direction now causes the room in that direction (immediate vicinity) to be attacked. I maintained the rooms in a graph class, which has them stored in an arraylist. For each room, however, it would have an associated BSTMap with its neighbors. In my game's case, each room has 4 neighbors, but it could also have as few as 1 or 2 if so desired. A method in the BSTMap class sets the edges between any two given rooms. For example, it can set room a to connect to room b vertically, so b's north neighbor is a and a's south neighbor is b.

Note that to represent directions, we used enums for the first time in class. This lets us predefine constants that a variable of that enum type can have. For example, a direction enum must be north, south, east, or west. The hunter in my game moves around by having his location defined by the room he is in, so every time we want him to move, we assign his associated room vertex to a new one. The wumpus, meanwhile, does not appear except after a win or a loss (his boolean for visibility starts off at false), and remains in its room. The hunter also sets the visibility of rooms he enters to true. The top level of the game, HuntTheWumpus.java, controls the game with its own enum, called PlayState, which can be PLAY or STOP. I used the control class, and had its keyTyped/keyPressed methods sense which key was pressed, then tell the hunter whether to move/equip a weapon, or tell the game to quit. Note that keyPressed is required for arrow key support. All of the logic follows the keystrokes. The main method was calling a method that told the thread to sleep, which determines how fast the events happen after pressing a button. I will discuss my code more below.

Class Explanations

The Hunter class has a field for the vertex it is in, whether or not the weapon is set, the image it is associated with, its width and height, and its dead or alive boolean. When deciding where to draw it, I get the (col, row) within the map of the vertex it is in, and I multiply both the col and row by the room width/height (either works because they are square rooms) to make it start at the right room. Then I add half of the room width or height for both the row and col of the image location, and I subtract half of the hunter width/height so that the middle of the hunter image is at the middle of the room. To handle the hunter image size, I ask how big the room is, and I define the hunter's size in relation to the room size. Because this results in a double, I use Math.round on the double and cast it to an int so we get the nearest whole number for the hunter dimensions (the drawing methods use ints, so we have to use ints). Each image associated with the hunter (normal hunter, equipped hunter, dead hunter) uses different dimensions. Here is an example of the code checking if the hunter is equipped:

```
else //if sword IS equipped
{
    double realWidth = roomWidth*.75;
    double realHeight = roomHeight*.75;
    this.imgWidth = (int) Math.round(realWidth);
    this.imgHeight = (int) Math.round(realHeight);
}
```

If the hunter is dead, we also move the hunter slightly down because it should look like he is on the floor, not in the center of the room standing. Note that the wumpus was extremely similar in structure to the hunter, and the only real difference it had was a method that checked if the room the attack entered is the same as the one the wumpus is in.

Some of the fields in Landscape include the window width and height, the list of foreground agents (hunter and wumpus), the list of background vertices (rooms), and the amount of rooms per row (or per column, because they are the same number). I had methods to get the wumpus and the hunter, which is where I had to loop through the foreground agent list and check if the agent was an instance of the class I wanted, then cast it to that object type when returning it. The landscape has the draw method, which calls the draw methods of each individual item in it. Note that when deciding how wide to make each room, we use the width/room line number, so that no matter how many room lines we have, it draws correctly. I also allowed the user to enter a number above 4 in the terminal to decide how many lines of rooms to have. Also, within the code, we can change the window dimensions, of course (as long as it remains square).

In LandscapeDisplay, I have fields for the landscape, the canvas panel, the width and height, and the images I use. In the constructor I try to read

in images with ImagoIO.read(new File...) and catch errors if they don't exist. There is also an update method, which creates a new graphics object by getting the canvas graphics, and draws everything on that, and also updates the hunter's room so it is visible. It also checks if the wumpus and hunter are in the same room, upon which it sets the wumpus to visible. For the winImage method, I check if the passed in win value is true, and I set the wumpus image to the defeated one if it is true, and otherwise I set the hunter image to the dead one and set the hunter's state to be dead. Either way, the wumpus becomes visible, as well. I also have a setBow method, which takes in a boolean for if the weapon should be set or not. If it should be (ie, true), the hunter's image will change and the hunter's state will change to reflect having a weapon set. If it is not true, it will set the image to the default unequipped hunter one, and the state will change accordingly. There is also a saveImage method in this class, as usual, which I used when making the gifs below; more on that soon.

The top level file, HuntTheWumpus, is where most of the logic behind the game flow is. I have a control class, as I mentioned, that uses an actionListener to pick up key presses. When making the gif, I created a variable that is incremented on each key press because every key press creates a new image. As an example, I will walk through what happens when the up key is pressed:

If the keyListener "hears" an up key (KeyEvent.VK_UP) and the northNeighbor of the hunter's current room is not null, an action will be taken. If the hunter's arrow is not set, the hunter's vertex will change to the north Neighbor. Then, the display will update. If the hunter is in the wumpus's room, the PlayState becomes STOP, and the main while loop is exited. Otherwise, it will continue. If the hunter's weapon had been set, it would have triggered the method I created in control called winOrLose, which calls the method in wumpus that checks if the vertices are the same. If they are, the winImage method of the display gets true passed in, and the game ends with the appropriate pictures. Otherwise, it gets false and the images are set accordingly. Either way, the state becomes STOP.

The last thing I want to discuss is how I built and connected all of the items in the map. Obviously, because I allow the user to specify how many rooms each side of the square window should have, the graph of room vertices is built dynamically. For this, I have a method called buildMap. It creates the links/edges between the rooms, sets the starting coordinates of the hunter and wumpus, and performs shortestPath on the wumpus's lair vertex at the end. I have a 2D array of all of the vertices, which I add two by having a nested for loop. From 0 to roomLines, each (column, row)-situated room vertex is created. Then I insert it into the landscape and the 2D array. After that, I loop over the 2D array in the same way, but now I want to decide how to connect them. Note that I decided to have the room connections wrap around, meaning you can walk off the edge of a side and end up at another side, like the farthest right to the leftmost side.

To do this, I ?had booleans that checked if the loop variables indicated that we were at the last column or last row. If we weren't on the last row, I would add an edge between the current vertex and the one south of it (row + ?1). Otherwise, I would add an edge between the current room and the first one in the column, meaning the row of that room should be 0. Similarly, if we were not on the last column, I would add a connection with the current vertex and the one to the east of it. If we were on the last column, I would simply make the connection between the current one and the one at (column 0, current row). Here is some of my code for that:

```
for (int c = 0; c < roomLines; c++)
{
    for (int r = 0; r < roomLines; r++)
    {
        //prepare for special cases
        boolean lastRow = false;
        boolean lastCol = false;
        if (r == roomLines-1) //last row
        {
            lastRow = true;
        }
        if (c == roomLines-1) //last col
        {
            lastCol = true;
        }

        //insert room vertex into array

        if (!lastRow) //not on last row
        {
            this.graph.addEdge(room2DArray[c][r],Vertex.Direction.SOUTH,
                               room2DArray[c][r+1]);
        }
        else //it IS on the last row
        {
            this.graph.addEdge(room2DArray[c][r],Vertex.Direction.SOUTH,
                               room2DArray[c][0]);
        }
```

After this, I put the wumpus at a random location, but checked if the column was 0, and did not allow its random row to be 0 as well if it was. Similarly, if it was not 0, I allowed the row to be 0. This would prevent the wumpus from ever landing on (0,0), which was where the hunter started. This is some of the code I used to check:
Then, I initialized the wumpus and hunter at their correct spots, and ran shortestPath on wumpus.This method is in the graph class, and it is based on Dijkstra's algorithm, which takes a given vertex and calculates the distance between that and each other vertex in the graph, updating the smallest distances as we go along. The ones within 2 spaces have the lowest "cost" to get to the wumpus, and those are told to be drawn with red walls in the vertex draw method.

Here are three gifs showing possible game results:

Success

Failure by entering Wumpus's lair

Failure by attacking incorrect room

Extensions

-Ext 1: Using BSTMap to keep track of neighbors

-Ext 3: Made the Wumpus appear in a different spot each time so each game is different

-Ext 5: Visually interesting (used images)

-Other ext: allowing user to specify rooms per line and dynamically changing map size and image sizes

-(Petty ext: I figured out how to use arrow keys to move and q to quit-- Does that count for anything?)

Conclusion

This project was the first time I used images*; I've always wanted to and never did, but I thought "If not now, then when?" when deciding to use them. It was very satisfying having them show up, and it was equally satisfying having the game respond to keystrokes, especially the arrow keys, because using WASD bothers me. This project taught me how to use enums, how to implement Dijkstra's algorithm, how to use and draw a BufferedImage, how to use the Control class, and how to represent infinity, among other things. It was also good practice in organization because of the number of classes, and I often found myself deleting redundant code that wanted to do something I'd already done somewhere else. Overall, it was a good experience, and I hope to make more games in the future, because this is what really interests me in CS.

Special Thanks: Melody, Stephanie

**Note that these base images are not mine, and I found them on the internet.*