Goblin movement function: optimalPath() & move() in Goblin.cpp.

Note: the optimalPath() function contain the entire algorithm that recursively finds the next optimal position for Goblins, and it is a helper function for move(); the move() function determines whether Goblins should move, attack, or do nothing and provides a 2D grid for the optimalPath() function.

The optimalPath() function starts from the current position of the goblin and checks all valid neighbors (up, down, left, right) using a depth-first search (DFS) approach, while keeping track of the current search depth. Here's how it works:

1. Initialization:
   * The function receives the current position of the goblin, the grid, the Temple object, the player's position, the current search depth (start from 1), and a reference to store the optimal next position.
2. Depth Check:
   * If the current search depth exceeds the goblin’s maximum smelling distance, the function returns -1, indicating that no valid path exists from this path.
3. Player Proximity Check:
   * If the player is in the neighbors of the current position, the function returns the current depth as it represents the distance from the goblin to the player via this path.
4. Recursive Search:
   * The function initializes minDistance to a value greater than the maximum smelling distance.
   * For each direction, the function checks if the position is navigable (empty space and not occupied by other actors).
   * It marks the position as visited.
   * Recursively calls optimalPath() to explore this direction, increasing the depth by 1, and update the minDistance.
   * After the recursive call, it unmarks the position to clear the way for the exploration other possible paths starting from neighboring directions.
5. Update Optimal Path:
   * If a valid path is found (i.e., minDistance is not its initial value and is not -1), the function will update the optimal Coord object, and the move() function will set it as the new position of the goblin.

The generation of rooms with corridors is divided into two functions. createRectangles() and generateCorridors(). Here is how they work:

createRectangles():

1. Generate Parameters:
   * Use the upper left corner as the anchoring parameter for a rectangular room.
   * Randomly generate the position of the upper left corner, as well as the width and height of the rectangle.
   * Repeat this process 9 times (an empirical value).
2. Check Overlap and Ensure Reasonable Number:
   * If the newly generated rectangle overlaps with an existing one, generate a new rectangle to avoid overlap.
   * Ensure the number of rectangles falls within a reasonable range (between 3 and 7).
3. Store Room Positions:
   * Store all positions of the rectangles into the vector m\_validCoords.
   * When generating other actors or objects, they will be placed at some position in the m\_validCoords vector.
   * When drawing the grid, positions not in the m\_validCoords vector are considered walls.

generateCorridors():

1. Initialization:
   * Determine the number of rectangles.
   * Create a corridorMatrix to track connections between rectangles, initializing it with zeros and setting the diagonal to one (indicating a rectangle is connected to itself).
2. Direction Tracking:
   * Initialize a vector triedDirection to keep track of attempted directions (up, down, left, right) for each rectangle.
3. Corridor Creation:
   * Iterate over each rectangle in the list.
   * For each rectangle, randomly select a direction (1 = top, 2 = bottom, 3 = left, 4 = right).
4. Direction Attempts:
   * Check if all directions have been tried for the current rectangle. If so, reset triedDirection and move on to the next rectangle.
   * Depending on the chosen direction, determine the starting point for the corridor.
5. Corridor Extension:
   * Depending on the chosen direction, extend the corridor from the starting point using helper functions (topCorridors, bottomCorridors, leftCorridors, rightCorridors) to determine the turn point and end point of the corridor.
6. Validation:
   * Check if the current direction works by verifying the turn point is valid and the corridor does not already exist between the rectangles.
   * If the direction fails, mark it as tried and retry with a different direction.
7. Corridor Construction:
   * If the direction is valid, create the corridor using appropriate helper functions (pushTop, pushBottom, pushLeft, pushRight).
   * Update the corridorMatrix to reflect the new connection between rectangles.
8. Repeat Process:
   * Reset triedDirection and continue the process for all rectangles until all possible connections are established.

Note: I understand that this way there might be duplicate value in the m\_validCoords vector, so there is extra procedure function to avoid initializing actors and objects on the same position. I wrote the program before Prof. Smallberg talked about set and unordered\_set. Had I known them, I would not use vector to hold the valid positions but rather a set to avoid duplicates.