CSC 584 HW4 Report

I decided to expose the following environment variables which would be used by the Sprite Decision Tree and the Monster Behavior Tree to make decisions.

1) Whether the sprite had entered a new room or not
2) Whether the sprite had finished ‘cycling’ the obstacle in the room
3) Whether the sprite and the monster were in the same room or not
4) Whether the monster had found and ‘eaten’ the sprite
5) Whether the monster was close to the sprite
6) Whether the monster had finished searching all the nodes in the room, looking for the sprite

All the above values were represented in boolean format and were used to control the behavior of the sprite and monster.

Decision Trees

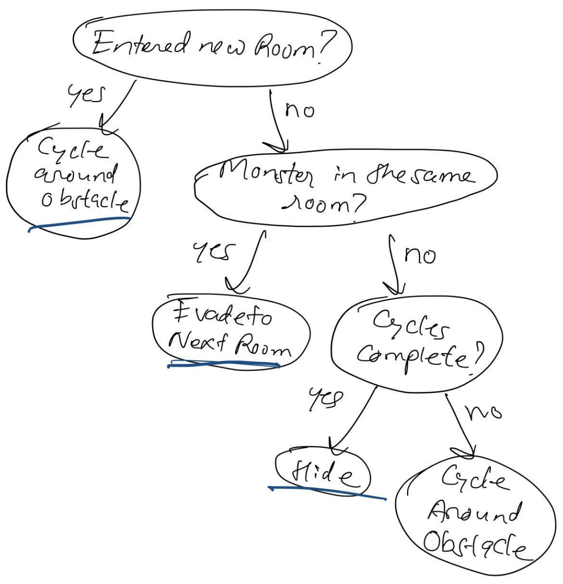
I created a custom data structure to store my Decision Tree using nodes of class **DTNode**. It is similar to a binary search tree in its structure. I had authored and inserted the nodes at different places as required. I also created a custom traversal method for the Decision Tree which would pass through the tree based on the values of the environment variables.

Description of Sprite Behavior -

If the sprite enters a new room, it must perform a ‘ritual’ or it must cycle around the obstacle present in the room exactly once. This number can be increased by tweaking one simple line present in the code. Once it has finished cycling the obstacle, it goes and ‘hides’ at the top left corner of the room. The sprite also has an emergency behavior - If it senses that the monster is in the same room, the sprite evades to the next available room. It then performs the ‘ritual’ of cycling the obstacle and then hides until the monster enters that room.

Note - ‘Cycling’ an obstacle is an example of following a predefined path. Evading to the next room or hiding use pathfinding to seek a certain point.

Below is an image of the decision tree which encodes this behavior.



Behavior Trees

Creating the Behavior Tree structure on paper wasn't too hard, but implementing the tree in code proved to be a far greater challenge than I had imagined.
A behavior tree stores state, meaning, it does not start its traversal from the root every time.

I created a custom data structure for my behavior tree, along with an algorithm for its traversal.

Following is an explanation of my behavior tree implementation -

I designed a pure virtual class called **BTNode** which I used as a template to create composite nodes (selectors and sequences) and tasks (for the different behaviors which sit at the leaf nodes of the tree). Each node in the tree has a **run()** function, which basically tries to evaluate the node (true or false). When the **run()** function in a composite node is called, it iterates over all its children and tries to evaluate each of them.

However, the problem arises when some nodes cannot be immediately evaluated (for example, task is evade - leaf node must wait until evade action is complete before returning true value). This is why each composite node (sequence or selector) and leaf node (actions or questions) can evaluate to three values - **0(false), 1(true), or 2(running)**.

We declare a global **deque<BTNode> runningNodes** to store the 'running' nodes.

If a leaf node evaluates to 2, it is added to the deque. If it evaluates to 0 or 1, it is not added.

If a composite node has one child that is 'running', its status also becomes 'running' and it is added to the deque.

Note that this behavior may scale up as one composite node may be the child of another composite node.

Hence at any point of time, the queue stores the list of nodes which are currently running (from the leaf node, all the way up to the root).

Now during the game loop, we pop off the last element from the queue (the leaf node) and try to evaluate it. If its status changed from **2** to **0 or 1** (as a result of changes in the environment variables) we pop the node from the queue and start executing the next node that is present (parent of the leaf node).

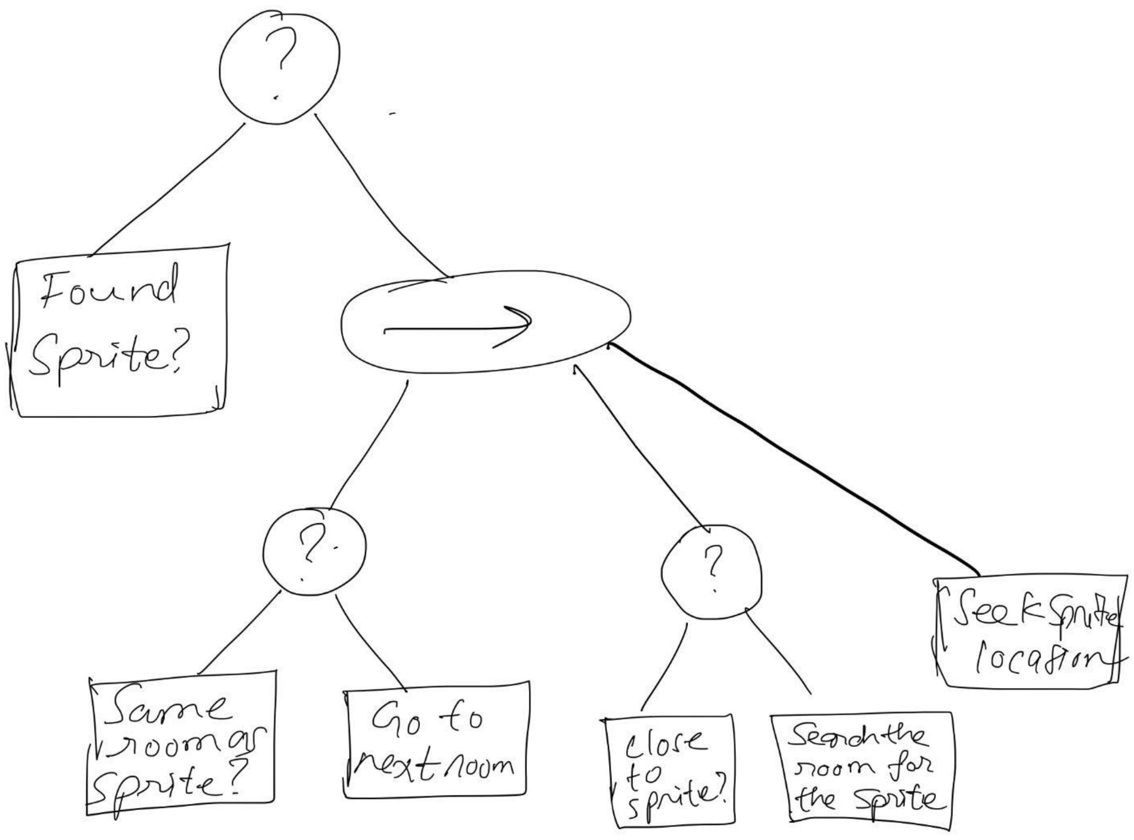
This way we keep adding ‘running’ nodes to the queue and execute the latest ‘running’ node until all nodes in the behavior tree have been executed.

Description of Monster Behavior -

The monster checks if it has already eaten the sprite. If it has then it does not need to do anything, its objective is already complete. If it has not eaten the sprite, it checks if it is in the same room as the sprite or not. If it is in the same room, it loops around the whole room looking for the sprite. If it sees the sprite in its FOV (a small circle around the monster) it seeks directly to the sprite location. If it succeeds in reaching the sprite and ‘eating’ it, the level resets and the monster and sprite respawn in their original locations. Going back to the second condition, if the monster does not sense the sprite in the same room it goes to the next room looking for the sprite. It keeps doing this until it finds the room where the sprite is and it begins searching in the room for the sprite and performs the behavior mentioned above.

Note - Searching for the sprite in the same room is an example of following a predefined path (the path cycles around the whole room). Moving to the next room is an example of pathfinding. Seeking the sprite location simply seeks to the sprite position (since it is very close to the sprite, it does not perform pathfinding any more).

Below is the diagram which represents the Behavior Tree -



Graphics -
The sprite is represented by the black sprite image from the previous homework.
The sprite leaves **red** breadcrumbs.
The monster is represented by a green monster image which has been downloaded from the internet.
The monster leaves **green** breadcrumbs.
The top left corner of the screen shows the DT and BT results along with the Sprite and Monster room numbers.

Decision Tree Learning -
Did not attempt this section.