CS7056 Autonomous Agents

Tom Moore 12307677

April 2, 2017

1 Introduction

For this project, we were assigned to create a gameboard modelling the Westworld environment described in Mat Buckland's "Programming Game AI by Example". This gameboard was to have autonomous agents that interacted with each other, based around a finite state machine implementation.
The project was divided into 4 broad based labs. Each of these labs contained tasks to be implemented to complete said lab. This project is the summation of those labs.
Those labs were:

1. Finite State Machines
2. Terrain Representation
3. Pathfinding
4. Sensing

As such my discussion of the project will us those labs as sections to describe my implementation.

2 Finite State Machines

In this first section I will discuss the finite state machine in use in my implementation, as well as the agents I implemented and their make-up. I will also discuss other features (such as messaging between agents) and my implementation of said features.

2.1 Add New Locations

For my project I chose to represent the important locations within Westworld, as an enum called "Locations". This enum contained all of the locations needed within Westworld. Later when I created my gameboard I saved the randomly generated coordinates of each important location to an array, with each location being stored at their instance of the enum.

2.2 Adding Bob and Elsa

I created Bob and Elsa as agents. Each individual agent is an extension of the agent generic. They were designed based around the following Finite State Machine Diagrams:

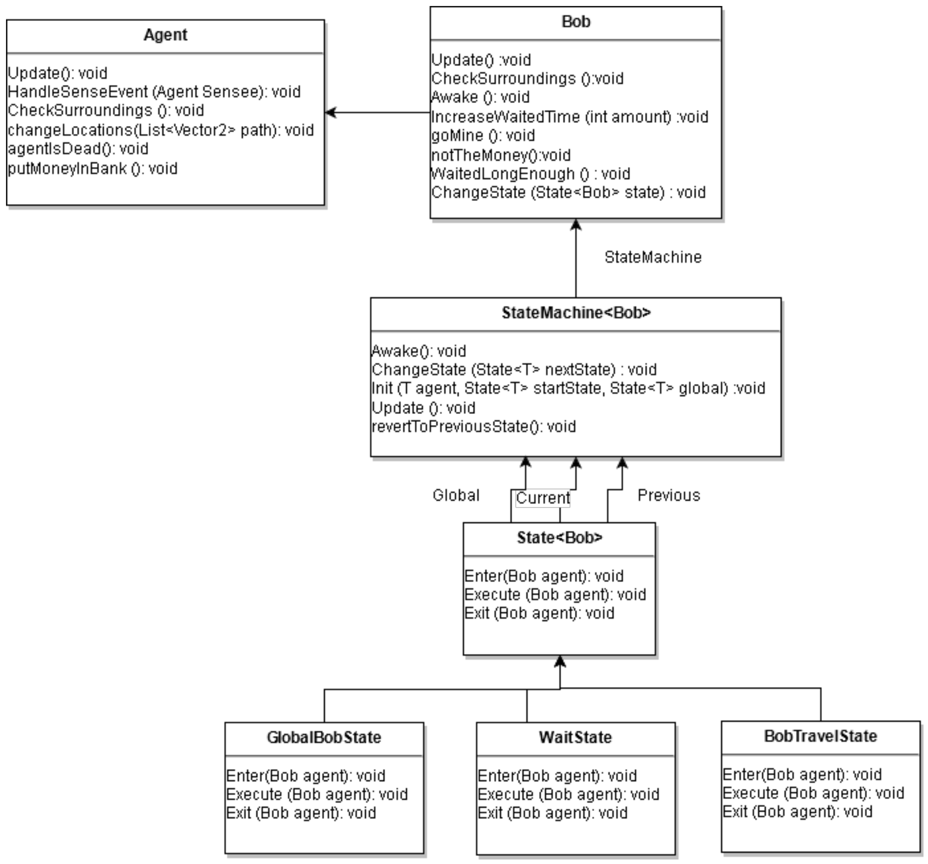
Agent
Update(): void
HandleSenseEvent (Agent Sensee): void
CheckSurroundings (): void
changeLocations(List<Vector2> path): void
agentIsDead(): void
putMoneyInBank (): void
Bob
Update() :void
CheckSurroundings ():void
Awake (): void
IncreaseWaitedTime (int amount) :void
goMine (): void
notTheMoney():void
WaitedLongEnough () : void
ChangeState (State<Bob> state) : void
StateMachine
StateMachine<Bob>
Awake(): void
ChangeState (State<T> nextState) : void
Init (T agent, State<T> startState, State<T> global) :void
Update (): void
revertToPreviousState(): void
Global
Current
Previous
State<Bob>
Enter(Bob agent): void
Execute (Bob agent): void
Exit (Bob agent): void
GlobalBobState
Enter(Bob agent): void
Execute (Bob agent): void
Exit (Bob agent): void
WaitState
Enter(Bob agent): void
Execute (Bob agent): void
Exit (Bob agent): void
BobTravelState
Enter(Bob agent): void
Execute (Bob agent): void
Exit (Bob agent): void

Figure 1: Bob Finite State Machine Diagram

Figure 1 represents the Bob Finite State Machine Diagram.

Figure 2 represents the Elsa Finite State Machine Diagram.

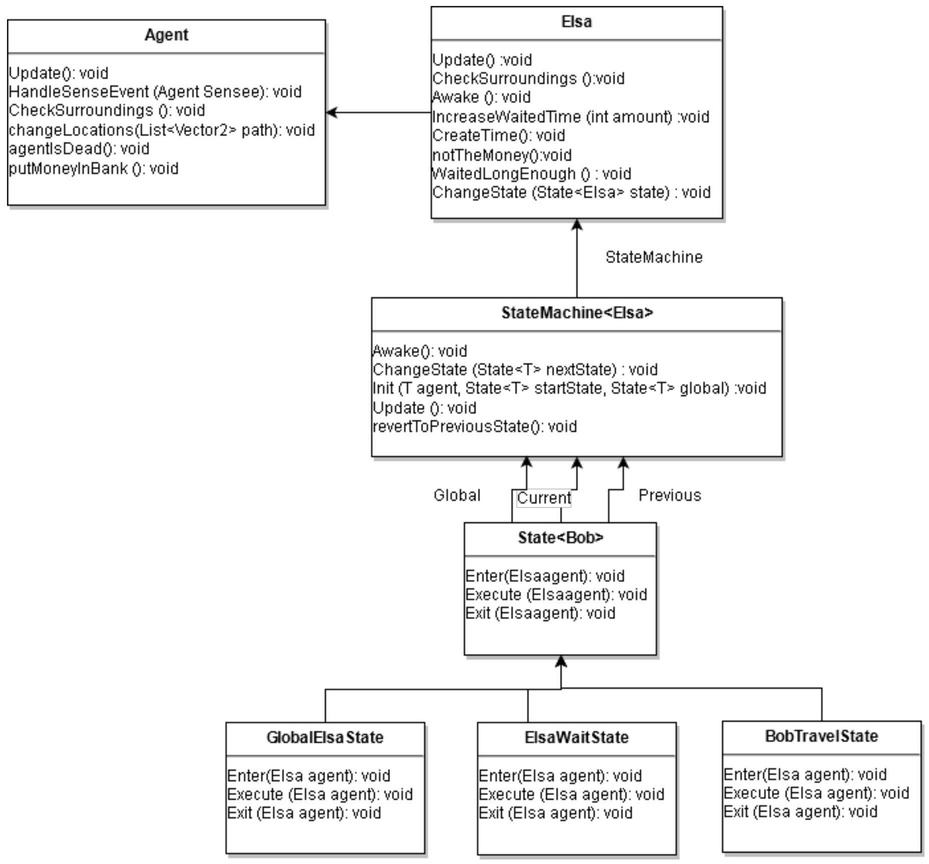
Agent
Update(): void
HandleSenseEvent (Agent Sensee): void
CheckSurroundings (): void
changeLocations(List<Vector2> path): void
agentIsDead(): void
putMoneyInBank (): void
Elsa
Update() :void
CheckSurroundings ():void
Awake (): void
IncreaseWaitedTime (int amount) :void
CreateTime(): void
notTheMoney():void
WaitedLongEnough () : void
ChangeState (State<Elsa> state) : void
StateMachine
StateMachine<Elsa>
Awake(): void
ChangeState (State<T> nextState) : void
Init (T agent, State<T> startState, State<T> global) :void
Update (): void
revertToPreviousState(): void
Global
Current
Previous
State<Bob>
Enter(Elsaagent): void
Execute (Elsaagent): void
Exit (Elsaagent): void
GlobalElsaState
Enter(Elsa agent): void
Execute (Elsa agent): void
Exit (Elsa agent): void
ElsaWaitState
Enter(Elsa agent): void
Execute (Elsa agent): void
Exit (Elsa agent): void
BobTravelState
Enter(Elsa agent): void
Execute (Elsa agent): void
Exit (Elsa agent): void

Figure 2: Elsa Finite State Machine Diagram

2.3 Add an Outlaw

My Outlaw can be represented best by his Finite State Machine Diagram which is shown if figure **??** My outlaw class contains methods that allow time to rob the bank and randomly choose a time to do so. They also allow for his death and respawning to occur. These happen by stopping the rendering for him till the undertaker reaches his body.He respawns when undertaker returns to the cemetery.

2.4 State Blips

I have implemented state blips by adding a global state that is called upon every, update along with the other state. This global state is used by every agent to trigger the sense function to see if there are other Agents near them. I have also implemented a return to previous state function that allows an agent to return

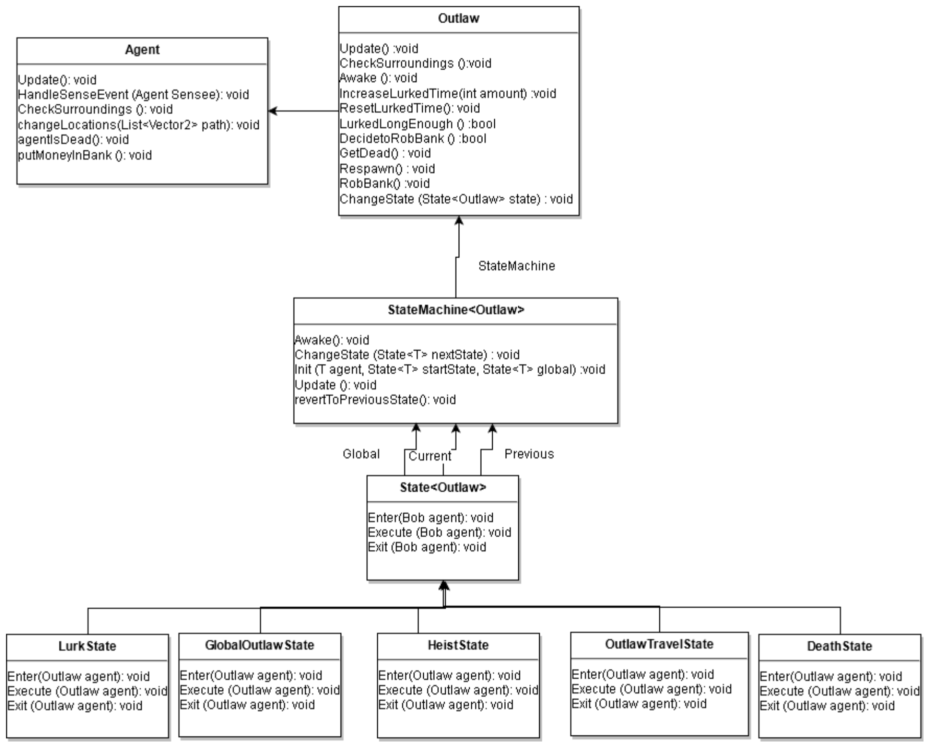
Agent
Update(): void
HandleSenseEvent (Agent Sensee): void
CheckSurroundings (): void
changeLocations(List<Vector2> path): void
agentIsDead(): void
putMoneyInBank (): void
Outlaw
Update() :void
CheckSurroundings ():void
Awake (): void
IncreaseLurkedTime(int amount) :void
ResetLurkedTime(): void
LurkedLongEnough () :bool
DecidetoRobBank () :bool
GetDead() : void
Respawn() : void
RobBank() :void
ChangeState (State<Outlaw> state) : void
StateMachine
StateMachine<Outlaw>
Awake(): void
ChangeState (State<T> nextState) : void
Init (T agent, State<T> startState, State<T> global) :void
Update (): void
revertToPreviousState(): void
Global
Current
Previous
State<Outlaw>
Enter(Bob agent): void
Execute (Bob agent): void
Exit (Bob agent): void
LurkState
Enter(Outlaw agent): void
Execute (Outlaw agent): void
Exit (Outlaw agent): void
GlobalOutlawState
Enter(Outlaw agent): void
Execute (Outlaw agent): void
Exit (Outlaw agent): void
HeistState
Enter(Outlaw agent): void
Execute (Outlaw agent): void
Exit (Outlaw agent): void
OutlawTravelState
Enter(Outlaw agent): void
Execute (Outlaw agent): void
Exit (Outlaw agent): void
DeathState
Enter(Outlaw agent): void
Execute (Outlaw agent): void
Exit (Outlaw agent): void

Figure 3: Outlaw Finite State Machine Diagram

to the previous state it was in.

2.5 Messaging

Messages are implemented via delegate and event pairs. There are two pairs. One that is held by the sheriff and triggered by the death of the outlaw, informing the undertaker to pick up the body. The other is held by the outlaw and informs Bob (or whoever else is subscribed) when he robs the bank.

2.6 Add a Sheriff

My Sheriff can be represented best by his Finite State Machine Diagram which is shown if figure **??** My Sheriff class contains methods that allow him to, kill an agent, and then message the Undertaker so that he is aware that said agent is dead, take an agents gold upon killing them, and then entering a serious of states that lead to the golds return. Finally he also has a function to randomly choose a new destination to patrol to, as long as it isn't the OutlawCamp.

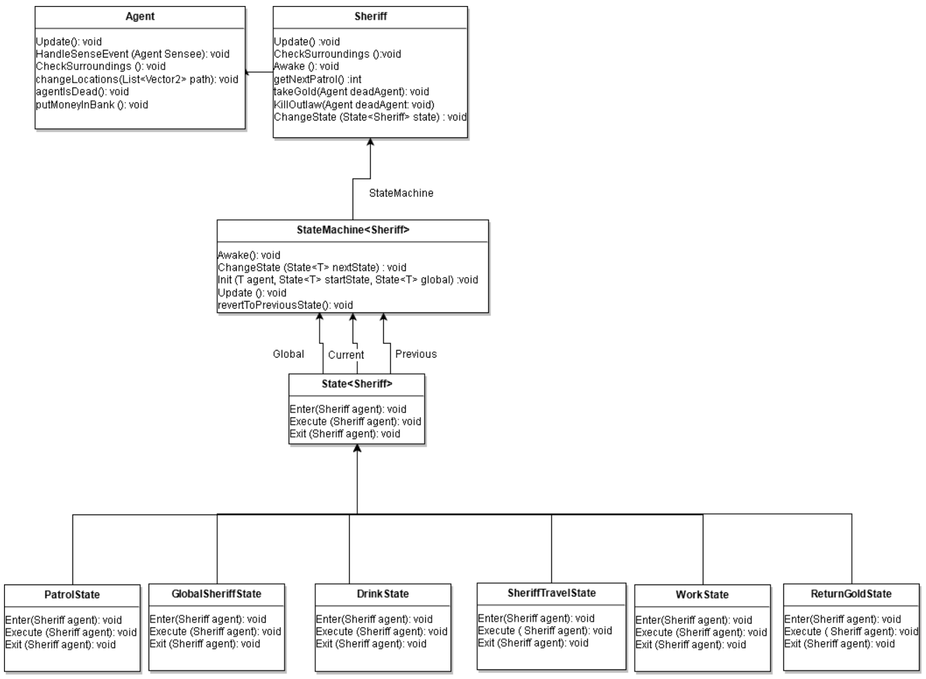
Agent
Update(): void
HandleSenseEvent (Agent Sensee): void
CheckSurroundings (): void
changeLocations(List<Vector2> path): void
agentIsDead(): void
putMoneyInBank (): void
Sheriff
Update() :void
CheckSurroundings ():void
Awake (): void
getNextPatrol() :int
takeGold(Agent deadAgent): void
KillOutlaw(Agent deadAgent: void)
ChangeState (State<Sheriff> state) : void
StateMachine
StateMachine<Sheriff>
Awake(): void
ChangeState (State<T> nextState) : void
Init (T agent, State<T> startState, State<T> global) :void
Update (): void
revertToPreviousState(): void
Global
Current
Previous
State<Sheriff>
Enter(Sheriff agent): void
Execute (Sheriff agent): void
Exit (Sheriff agent): void
PatrolState
Enter(Sheriff agent): void
Execute (Sheriff agent): void
Exit (Sheriff agent): void
GlobalSheriffState
Enter(Sheriff agent): void
Execute (Sheriff agent): void
Exit (Sheriff agent): void
DrinkState
Enter(Sheriff agent): void
Execute (Sheriff agent): void
Exit (Sheriff agent): void
SheriffTravelState
Enter(Sheriff agent): void
Execute (Sheriff agent): void
Exit (Sheriff agent): void
WorkState
Enter(Sheriff agent): void
Execute (Sheriff agent): void
Exit (Sheriff agent): void
ReturnGoldState
Enter(Sheriff agent): void
Execute (Sheriff agent): void
Exit (Sheriff agent): void

Figure 4: Sheriff Finite State Machine Diagram

2.7 Cleaning Up the Corpses

My Undertaker can be represented best by his Finite State Machine Diagram which is shown if figure **??** My Undertaker class contains methods that allow him to, retrieve the dead outlaws body and check if anyone's dead. He receives a message from the sheriff upon the outlaws death at the sheriffs hands. He then goes to the outlaws body and retrieves, allowing the outlaw to spawn. He then returns to hovering in his office.

3 Terrain Representation

In this section I will discuss my implementation of a gameboard to show the terrain and agents. This will include my implementation of locations, as well as generating a random gameboard.
It should be noted that I chose to follow the 2D RogueLike Tutorial for the creation of my gameboard, which represents my game.

3.1 Add Geography to West World

To add geography to Westworld, within my BoardManager I created two new gameobjects (plains and mountain). The plains act as the base floor for the

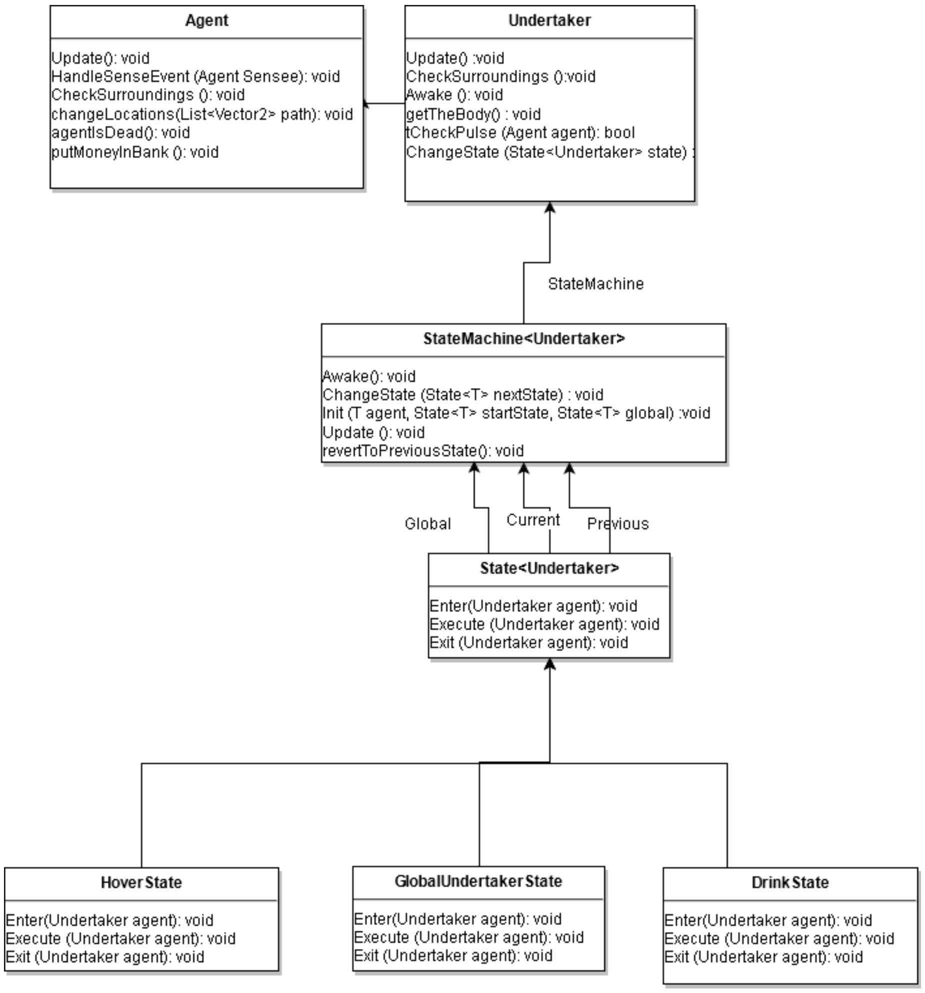
Agent
Update(): void
HandleSenseEvent (Agent Sensee): void
CheckSurroundings (): void
changeLocations(List<Vector2> path): void
agentIsDead(): void
putMoneyInBank (): void
Undertaker
Update() :void
CheckSurroundings ():void
Awake (): void
getTheBody() : void
tCheckPulse (Agent agent): bool
ChangeState (State<Undertaker> state) :
StateMachine
StateMachine<Undertaker>
Awake(): void
ChangeState (State<T> nextState) : void
Init (T agent, State<T> startState, State<T> global) :void
Update (): void
revertToPreviousState(): void
Global
Current
Previous
State<Undertaker>
Enter(Undertaker agent): void
Execute (Undertaker agent): void
Exit (Undertaker agent): void
Hover State
Enter(Undertaker agent): void
Execute (Undertaker agent): void
Exit (Undertaker agent): void
GlobalUndertaker State
Enter(Undertaker agent): void
Execute (Undertaker agent): void
Exit (Undertaker agent): void
Drink State
Enter(Undertaker agent): void
Execute (Undertaker agent): void
Exit (Undertaker agent): void

Figure 5: Sheriff Finite State Machine Diagram

entire board, with them being placed to the dimensions of the board. The mountains are placed at random, based on the LayoutObjectAtRandom function which takes a list of gameobjects and places them at random positions.Mountains much like all of the Westworld locations are layered as ”items”. This is a higher layer than ”floor”. As such they are rendered on top of the plains.

3.2 Add the West World Locations

To add my Westworld locations, I created several gameobjects, one for each location. I then instantiated them to a random position. The position of each gameobject was stored in a public array, where the index of each location was there Locations enum.

3.3 Drawing West World

I added prefabs to each instantiated game object, within my BoardManager. Each prefab contained a sprite that was rendered at a particular layer. The layer for the plains prefab was lower than the lay for all of the other prefabs, ensuring it would be rendered below any other gameobject.

3.4 Drawing Agents

I created and instantiated several gameobjects, one for each of the agents. Each agent was given a prefab that contained the agent's script. As well as a sprite representing them. As they were an easy set of sprites to use, I chose to use the 2D Roguelike Tutorial sprites for all of my tiles and agents. The prefabs are shown in figure [?]. An example of my Gameboard is shown in figure [?].

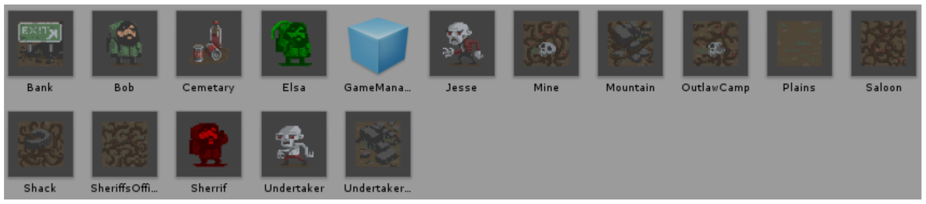

Figure 6: The prefabs and their sprites.

4 Pathfinding

In this section I will discuss my implementation of the A* pathfinding algorithm, as well as examples from my implementation.

4.1 Implement a Suitable Data Structure

I created two data structures for use in using the A* pathfinding algorithm. The first was a node structure, that contained, the x and y co-ordinates of the node on a grid, as well as an associated penalty to cross said node. Each node, also had an associated "g" and "h" cost, for implementation with my A* algorithm.Each node was contained within a grid of nodes, of a fixed width and height. Each of grid was able to retrieve the neighbours of a particular node.

4.2 Implement A*

I implemented the A* algorithm using a lazy diagonal heuristic. Each node had an associated g and h cost which were combined together to retrieve an f cost.

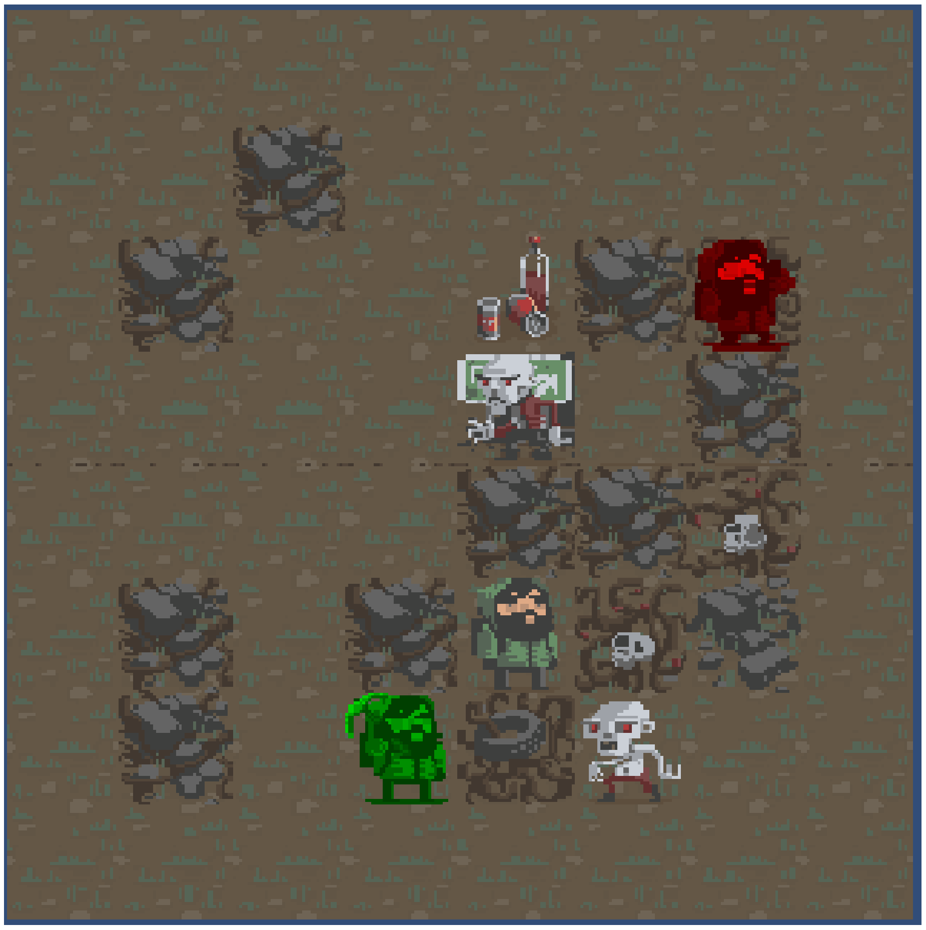

Figure 7: An example of my GameBoard in action.

The g cost is the cost of the path from the starting point to any that node. The h cost is the heuristic estimated cost from that node to the goal.

4.3 Make the Agents Move

To make my agents move I created a grid within my BoardManager, based off the board, where each node had a the position of a tile, and the cost associated with travelling across that tile. Then whenever an agent enters a travel state, it takes with them the destination it wishes to go to. In that state the agent uses the grid to generate a path of easiest movement. In my generic agent class, each agent takes the position of the next node on the list of locations return as the path, until they arrive at the destination.

5 Sensing

In this section I will discuss the rudimentary sensory capabilities that were added to my agents such that they were able to sense events in my game world.

5.1 Add a Sense Events and Sense Handlers

In the global state of each agent I have a call to sense the surroundings to see if there are any other agents nearby. This is done by the senseEvent class which contains an enum to represent three senses, sight, hearing and smell.
It also has a sense function that implements the sense of sight. It does this by checking the distance between the agent and agent being sensed. If the agent is close enough they are sensed and that sensing is handles by a function in each agent class.

5.2 Adding Sensory Attenuation Data to Your Map

Sensory attenuation data was added to my map in the same way as I added the travel costs. In added a sight smell and hearing propagation cost to each tile, by storing said info in an array of loats containing the location of each tile. This data could then be added to a grid for use with A* by substituting the tile cots for the propagation costs.

5.3 Other Senses

Unfortunately I was unable to correctly implement smell, and hearing as senses in my game.

5.4 Sensory Perceptiveness

I would add sensory perceptiveness to my agents by having a perceptiveness float that could be modified according to the situation to each agent. This float would be used as a modifier on the sensory propagation data.