Bob The Miner
Implementation Report

Samir Kumar 16341316
08 April 2017

1 INTRODUCTION

This report consists information about the implementation of autonomous agent based game AI. This game's scenario is set in 2d tile based world. It has multiple agents with their different behaviours. The name of this game is based on the couple in the game Bob and Elsa. Bob is the character in the game who works in the mine. The world implemented in this game is based on description of Buckland West World.

This implementation mainly consists of four vital elements. These four factors are Finite state machines, terrain representation, path finding and sensing. These four elements and their implementation is further discussed in the report.

2 IMPLEMENTATION

2.1 FINITE STATE MACHINES

Finite state machines start up code was provided. It included Bob as an agent with some of his behavioural tasks. Agent and state as abstract base class.

State class is made to have three functions enter (), execute () and exit (). This base class becomes reusable like a template. It takes agent as object parameter and supports it to switch between states.

Per the first task in lab 1, Bob agent is created as a new game object. Second task, is to learn about Generic as they are used vitally in the project. To fulfill the third task, state machine class is updated to have all the functions from agent class. Agent class is left with only update () function which evokes the update function of each agent's own state machine.

Agent object is passed through the state machine class for each agent. When agent gets active, it starts in its enter state. If this enter state exists then execute it, else do nothing. Change state () function allows the agent to go to next stage if it exists. If next state doesn't exist it go to exit state.

Task 4 was to create 4 locations, Outlaw Camp, Sheriffs Office, Undertakers, Cemetery. enum is used as shown in code listing 1

```
public enum Locations
{
        None,
        Cemetary,
        OutlawCamp,
        UndertakersOffice,
        sheriffOffice
}
```

Listing 1: Code representation: Locations enum consisting all locations

State Blips:
Task 6 was done before task 5. It sounded reasonable to setup global state and state blips before creating more agents. To perform state blips, current state was stored in the previous state variable before changing. It kept the record of previous state to which agent can return.

Another function reverttopreviousstate () was made to perform blips. As shown in code listing 2

```
//change state back to the previous state
        public void RevertToPreviousState()
        {
                ChangeState(this.previousState);
        }
```

Listing 2: Code representation of reverting to previous state function

Now every agent should have global state which is updated continuously. Agent stays in global state unless it is triggered to change state using changestate () function. If agent is finished doing its blip, then is reverts to the previous state using function shown above.

This way every agent have their own state machine and can change states easily.

So, agent Bob now just specifies its start state and global state its bob class. Rest is handled by Bob's state machine. Example is shown in code listing 4,

```
protected override void Start()
    {
                base.Start ();
                this.stateMachine = new StateMachine<Bob>();
                this.stateMachine.Init(this, EnterMineAndDigForNugget.Instance);

        InvokeRepeating("UpdateStateMachine", 1, 1);
    }
```

Listing 3: Code representation start function after state machine implementation

Task 5 was creating outlaw agent. It was created in similar way as bob agent. To get his required behaviour, another location was added called bank. Outlaw agent starts at location outlaw camp and lurkaround state is assigned by its global state which is updated continuously.

In lurkaround state, outlaw agent is moving from outlaw camp to cemetery and back. After random

number of cycles in this state, it changes state to another state of robbing bank. After robbing bank, it reverts itself to its previous state of lurkaround state.

Messaging:
Task 7 was messaging. Now, we know that all our agents print message while they do something in debug log. Messaging is like when some event happens it is broadcasted to the agents in relation to that event. So, communication between two agents is also considered as messaging.

Agent Elsa was introduced as Bob's wife. So, that they can do messaging between each other. Bob Elsa messaging is probably more obvious it is still implemented and can be seen in the code. Instead in the report, messaging between Sheriff, Undertaker and Outlaw is being explained. After basic messaging system implementation more complex implementation is organized after last lab with sensing. It is still a better example to explain the functionality of messaging.

To implement messaging system, first agent class for all agents is used to declare a bool variable to handle message. Telegram class was made which contains content type of the messages being dispatched and handled. Telegram has enum of all the messages used in the world. Struct is used to define content of a telegram. Sender, receiver, message type and despatched time are the variable essential for a telegram.

Message dispatcher class handles all the messages. It handles the message to be dispatched immediately or after some time using delay. Function dipatchmessage () is used for both purposes. When one agents send message to another agent this function is called and executed. It contains telegram information such as sender id, receiver id, message itself and time delay. For immediate messages delay = 0.

If the message is delayed it is stored in the que. Dispatchdelyaedmessages () function handles delayed messages on priority basis. Once the message is dispatched it is removed from the front of the que.

Now the actual messaging interaction between Sheriff, Undertaker and Outlaw.
If outlaw is sensed by sheriff while in his global state. Message “Sheriffencountered“ is broad casted by Sheriff.
On this message Outlaw reacts with log saying “shot by sheriff“.
Outlaw state is changed to DeadOutlaw.
This state dispatches a message “OutlawDied“’ to Undertaker.
Undertaker handles this message and goes to the location where outlaw is shot.
This again triggers another message of “UndertakerArrived“’ to outlaw agent.
It executes the function hidebody (), which sets outlaw agent inactive. (that means undertaker takes his body)
Undertaker then goes to location cemetery with Outlaw's body. There it dispatches message for outlaw to re spawn.

This describes the functionality of messaging between agents. Figure no. 2.1 shows the Debug Log of these messages. They can also be seen on the game screen as shown in figure 2.2.

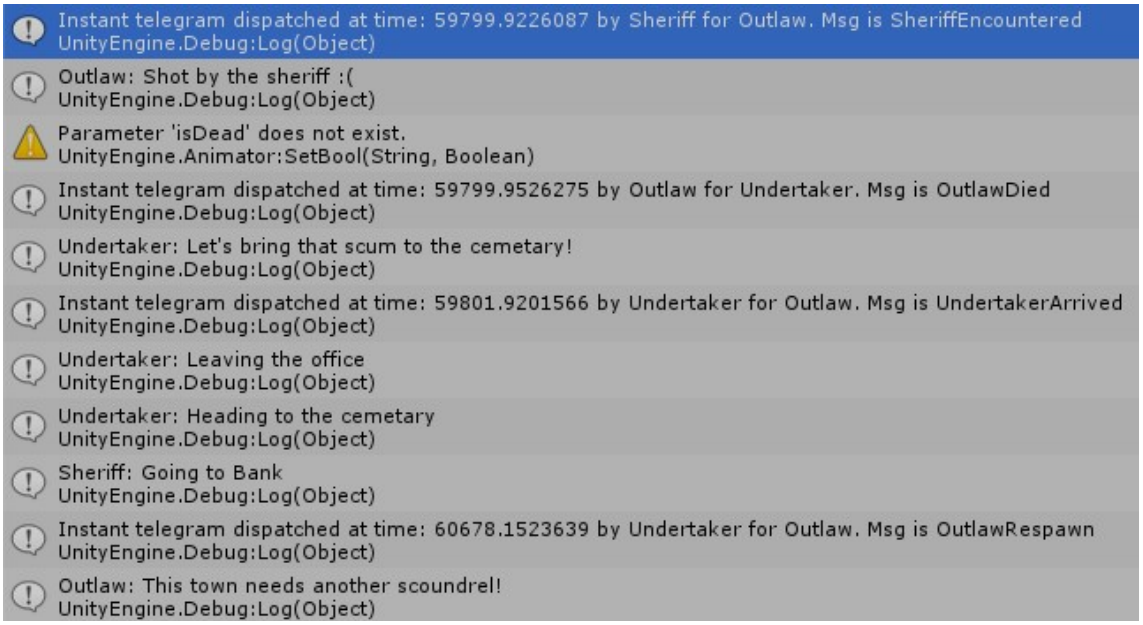

Figure 2.1: Debug Log showing messaging functionality

Figure 2.2: Terrain Representation with Messaging

Task 8 adding Sheriff to the scene is almost explained above. To make sheriff patrol all locations one by one except Outlaw Camp which he will never be able to find. Outlaw camp was put on number 9 location. Sheriff was not allowed to go above location 8. As can be seen from code listing 4 below.

```
var values = Enum.GetValues(typeof(Locations));
if ((int)agent.location < 8) {
var nextLocation = (Locations)(((int)agent.location + 1) %
values.Length);
            Debug.Log(agent.ID + ": Going to " + nextLocation);
            agent.ChangeLocation(nextLocation);
        }
        if ((int)agent.location == 8)
        {            var nextLocation = (Locations)
        (((int)agent.location - 7) % values.Length);
            Debug.Log(agent.ID + ": Going to " + nextLocation);
            agent.ChangeLocation(nextLocation);
        }
```

Listing 4: Code representation for Sheriff to travel all locations except Outlaw camp

Task 9 implementation is mostly covered above.

2.2 Terrain Representation

Lab 2 Terrain Representation is about giving this world a visual look. Unity 2D Roguelike tutorial was used to set up the tile based environment. 8 by 8 rows and columns were used to generate map. RandomPositision () function was used to randomise the position of each object. As shown in listing 5 below

```
//RandomPosition returns a random position from our list gridPositions.
Vector3 RandomPosition ()
{
//Declare an integer randomIndex, set it's value to a random number
//between 0 and the count of items in our List gridPositions.
int randomIndex = Random.Range (0, gridPositions.Count);

//Declare a variable of type Vector3 called randomPosition, set it's
//value to the entry at randomIndex from our List gridPositions.
Vector3 randomPosition = gridPositions[randomIndex];

//Remove the entry at randomIndex from the list so that it can't be
//re-used.
gridPositions.RemoveAt (randomIndex);

//Return the randomly selected Vector3 position.
return randomPosition;
}
```

Listing 5: Code representation to randomize grid and location

All the location per west world environment were added to the map. Each object has its own prefab. Forest, outer walls, floor tiles and all the locations from location list were the added. Boardmanager.cs file contains all the mapping information. Free sprites were downloaded from Internet and were used in unity.

2.3 Path Finding

A* algorithm has been implemented for path finding for agents. Node class consist of all the node information. Data structure for the grid has already been made using x, y node addresses.

To implement A star, a function called calculatepath () is implemented. It consists of all the information such as starting node, ending node, open nodes and closed nodes.

Neighbor function is used to calculate distance using Manhattan Distance. This function checks all the neighboring nodes in all for direction. It checks which node is assessable and can walk or not. It always uses straight left, right, up and down movements and no diagonal movements are performed.Manhattan Distance calculation is shown in code listing 7. neighbour() and other functions are in aStar.cs class.

```
    public int ManhattanDistance(Point point, Node goal)
    {   // linear movement - no diagonals - just cardinal directions (NSEW)
        return Math.Abs(point.x - goal.x) + Math.Abs(point.y - goal.y);
    }
```

Listing 6: Code representation Manhattan Distance Calculation

Max walkable tile number is 0. As shown in code listing 7. Anything above this number is unavailable to travel for the agent. We try to search the path which is most likely to lead towards the goal. We keep adding the nodes for cost calculation. When the open set of nodes is empty then we know that this is the optimal shortest low cost path. That path is the result array of nodes.

```
const int maxWalkableTileNum = 0;

    // returns boolean value (world cell is available and open)
    private bool canWalkHere(int x, int y)
    {
        return (world[x, y] <= maxWalkableTileNum);
    }
```

Listing 7: Code representation to randomize grid and location

Open array and closed list is used to make the search more efficient. We take the distance in to account as node g is the cost property from the starting point. f property is g with addition to cost estimated from start to destination. Lower f value of a node has the higher priority. With every step, one of the lowest f value is removed from the queue and f value and g value of neighbor nodes are updated. So, they are nest nodes to be traversed. We continue until nodes f value of lowest of a node from all other nodes.

Tile movement cost given to forest and mountains is then added on the path cost calculation. So that agent can avoid to go over these is its not as expensive.

Smooth movement and moving agent code is in movingObject and movingAgent class respectively. Smooth movement helps the agents to go from node to node rather than teleporting from one place to another.

2.3.1 SENSING

Agent class is used to provide agents with declaration of handling sense events. Sense class is made for the sensing function. To start with enum is used to declare the type of senses. Three senses are declared in this hearing, sight and smell.
Hearing:
First sense implemented is hearing. It takes the position of all the agents and check their position to all other agents except themselves. If the position is with in the hearing distance than they can sense each other. As can be seen in the code listing 8 below. If the path between them exists according to aStar algorithm then sensing is in effect.

```
// If close enough
if (Vector2.Distance(a1.CurrentPosition, a2.CurrentPosition) < SENSE_RANGE)
{
// Propogate the sense
var a1Pos = new Point() { x = (int)a1.CurrentPosition.x, y =
(int)a1.CurrentPosition.y };
var a2Pos = new Point() { x = (int)a2.CurrentPosition.x, y =
(int)a2.CurrentPosition.y };
if (aStar.calculatePath(a1Pos, a2Pos) != null)
{
// Sense the agent
Sense sense = new Sense(a2.ID, a1.ID, SenseType.Hearing);
a1.HandleSenseEvent(sense);
}
```

Listing 8: Code representation Hearing sense according to position

To explain the handling of hearing sense event, example of sheriff sensing outlaw is used in the report.Now sheriff handles the sense event. If sheriff senses outlaw within the range of hearing distance, then he shoots outlaw (i.e. message is dispatched). Then any action can be executed such as message is dispatched saying sheriff encountered. As shown in code listing 9.

```
public override bool OnSenseEvent(Sheriff agent, Sense sense)
    {
if (sense.Sender == "Outlaw" &&
!(EntityManager.GetEntity(sense.Sender) as Outlaw).isDead)
{
MessageDispatcher.DispatchMessage(0, "Outlaw", agent.ID,
MessageTypes.SheriffEncountered);
return true;
    }
 return false;
    }
```

Listing 9: Code representation: Sheriff handling hearing sense event

Sight:
Similarly, second sense implemented is Sight. Physics2d.OverlapCirlce is used to detect other agents around each agent. This is again implemented globally. Ray casting only gives back the game object. So,

to pass it through to all agents new struct had to be constructed. Below is the function code listing 10

```
public static void sight()
{

        for (int i = 0; i < EntityManager.GetCount(); ++i)
        {
            Agent a1 = EntityManager.GetEntity(i);

agents = Physics2D.OverlapCircleAll(a1.transform.position, sightrange,
1 << LayerMask.NameToLayer("Agents"));
            foreach (Collider2D c in agents)
                    {
                    Sense2 sense2 = new Sense2(c.gameObject, a1.ID,
                    SenseType.Sight);
                     a1.HandleSenseEvent2(sense2);
                    }

                }
           }
```

Listing 10: Code representation: Sight Sense Implementation using Ray cast

Ray casting to all the objects is expensive. To get efficiency out of it, it is only limited to the layer called agents. So, list of all agents check their ray casting to all other agents only, not with walls, Forrest or building etc unless we want them to. It was only possible because all the agents have a 2d box Collider.

Execution of this sensing event can be seen in an example to sheriff sensing undertaker with in the sight range of 3. Range of sight is kept more than hearing for realism. If he sees undertake it simply logs 2 messages. First message tells which agent sensed which agent by which sense. Second simply a greeting from Sheriff the who sensed Outlaw. Code listing 11 is shown below as example

```
    public override bool OnSenseEvent2(Sheriff agent, Sense2 sense2)
    {
        if (sense2.Sender == GameObject.FindWithTag("Undertaker"))
        {
            Debug.Log(agent.ID + " Saw " + sense2.Sender + " By
            Sense:" + sense2.senseType);
            Debug.Log(agent.ID +": Hope You Doing Your Job Properly "
            + sense2.Sender);
            return true;
        }
        return false;
    }
```

Listing 11: Code representation: Sheriff handling sight sense event

Below the image no. 2.3 shows debug log printed after sensing by sight event. Then next is the image of physical world where sight sensing happened in figure no. 2.4.

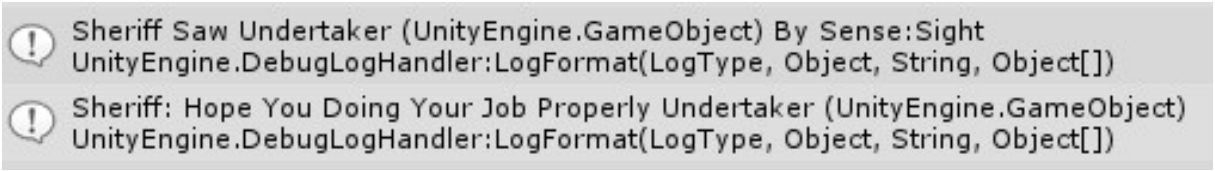

Figure 2.3: Debug Log as an action of sight sensing Displaying all information about sender, receiver and sense type.

Figure 2.4: Physical World Representation when sight sensing happened.

3 Future Work

There is a lot of future work that can be done in the implementation with having more time. First, the grid can be transformed in to 3D. A star algorithm can be improved to work more efficiently. Sensing Can be improved by adding obstacles in sight and more senses with complexity. Sensing can be also organized in a slightly better way from code refactoring regards.

4 References

[1] Christer Kaitila(2013) A-STAR Pathfinding AI for HTML5 Canvas Games.Availabe at: http://buildnewgames.com/astar/
[2] Mads Haahr (2017) CS7056-A-Y-201617 (Autonomous Agents) Lecture Notes.
[3] 2D Roguelike tutorial- Unity Tutorials
[4]Mat Buckland (2005). Programming Game AI by Example. (online) Available at: `https://books.google.ie/books/about/Programming_Game_AI_by_Example.html?id=gDLpyWtFacYC&redir_esc=y` (Accessed 2 Apr. 2017).