COSC 2527 Games and Artificial Intelligence Techniques Report

Semester 2, 2019

(Assignment 1 – Group 5)

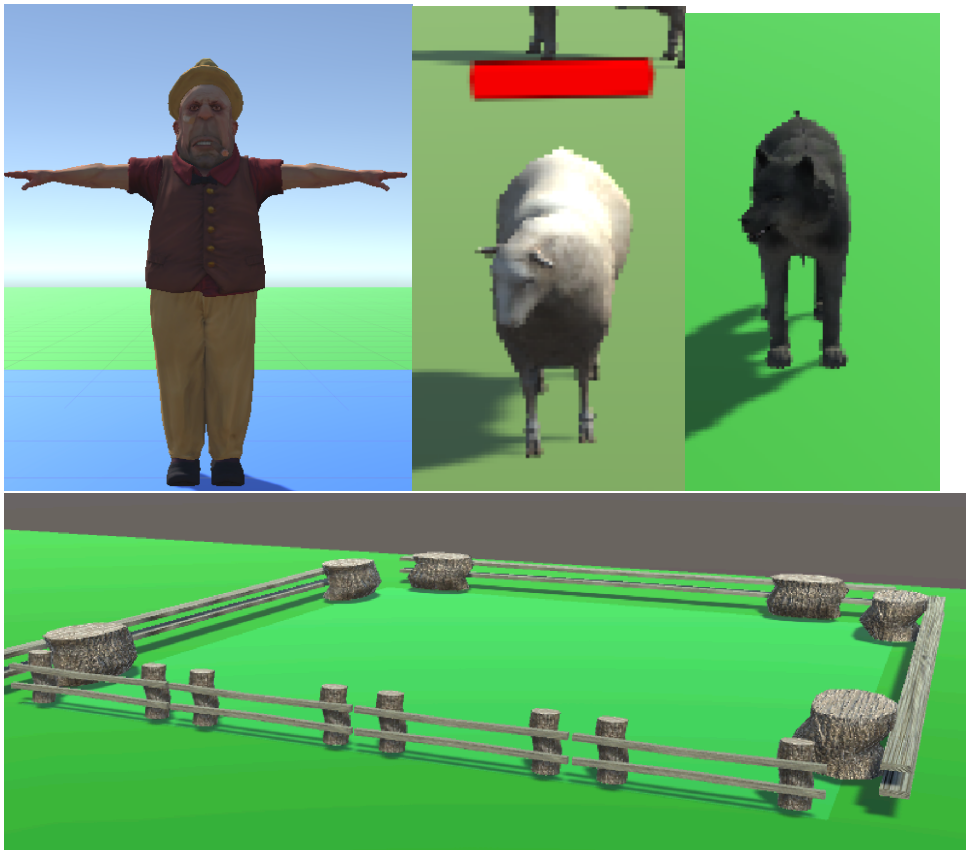
Report Date	30/08/2019
Group Members	Mads Bjørn(s3799147) Haixiao Dai (s3678322) Yifan Wang(s3672150)

1 Introduction

1.1 What is our game:

You, the player, controls the framer. The farmers goal is to keep the sheep alive for a long as possible, there is no wining only losing slowly. The sheep can die in two ways, the first way is to be eaten by the wolf, and the second is starvation. The sheep start in fenced of areas where the wolf cannot get to them, but they cannot stay there since the grass in the fenced of areas limited. The farmer can move the sheep between multiple fence of areas, to make sure that have enough food, but while you move the sheep the wolf will try to get them. When there is no sheep in a fenced of area, the grass in that areas will regrow.

Below is the farmer, sheep, wolf, and a fenced of areas illustrated:



1.2 How to play it (controls):

The farmer is controlled by the arrow keys / WASD, the up and down key is forward and backwards, while the left and right keys rotate the farmer counter-clockwise and clockwise.

To open and close the fenced of areas, click on a fence with the mouse, this will make the fence go down, and after 4 seconds it will go up again if there is not sheep or farmer on top of the fence.

To control the sheep, you must take advantage of the fact that the sheep try to flock together, and the sheep try to keep a distance to the farmer. This can be used to “push” the sheep slowly forward.

Camera control: we have two cameras one from the top down, and one from the perspective of the farmer. You can switch between the cameras with the space button and zoom in / out with the scroll wheel of the mouse.

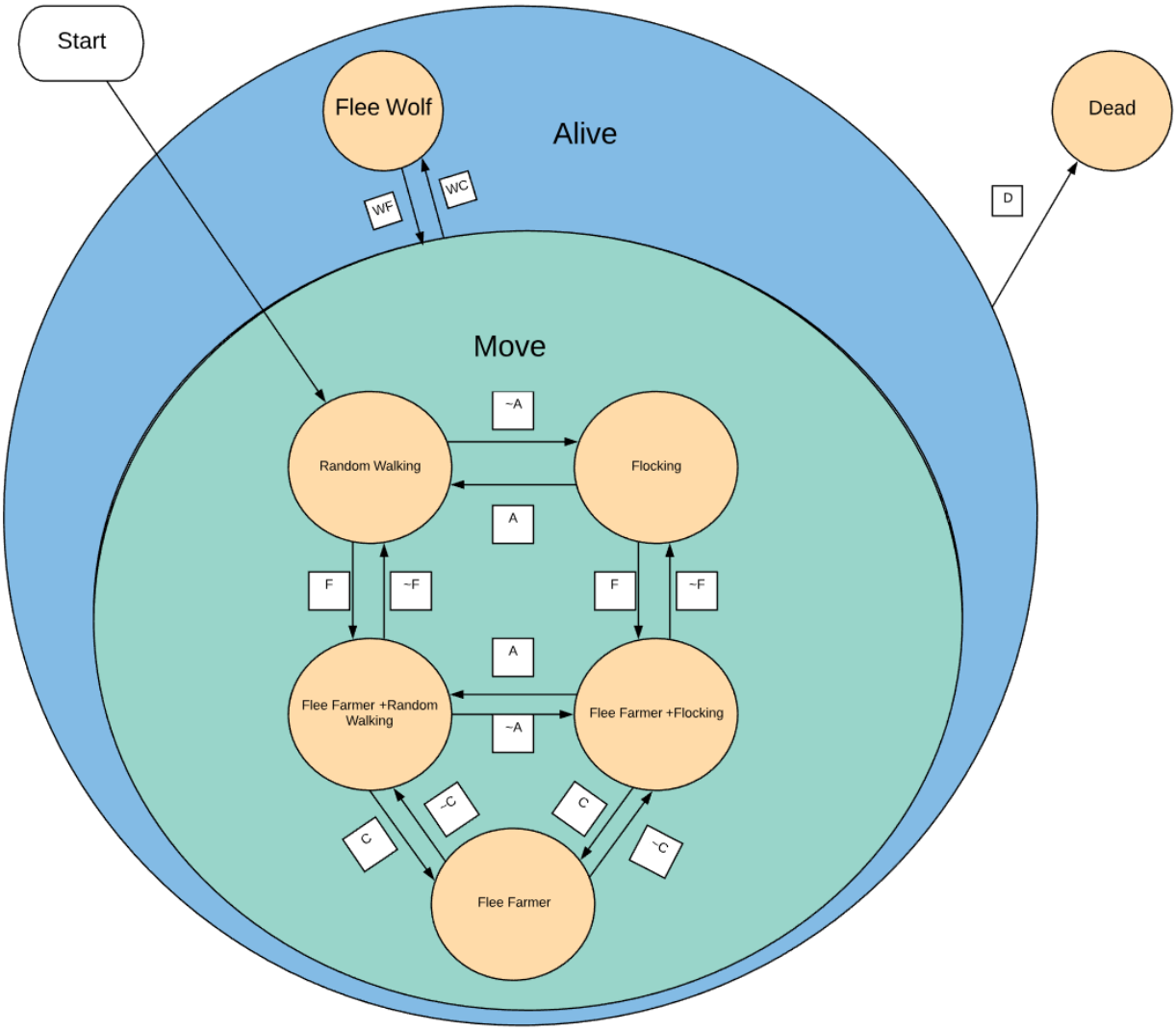
Main Scene: Scenes/SampleScene
Show case of pathfinding and steering: Scenes/PathFindingExample

2 Algorithms

2.1 Finite State Machine

2.1.1 Design the FSM diagram:

- For the sheep:



Events:
A : alone
F : farmer close
D : dead
WC: wolf close (<=10)
C : farmer closer
WF: wolf far away (>=15)

The details of each states are described in “Flocking” part.

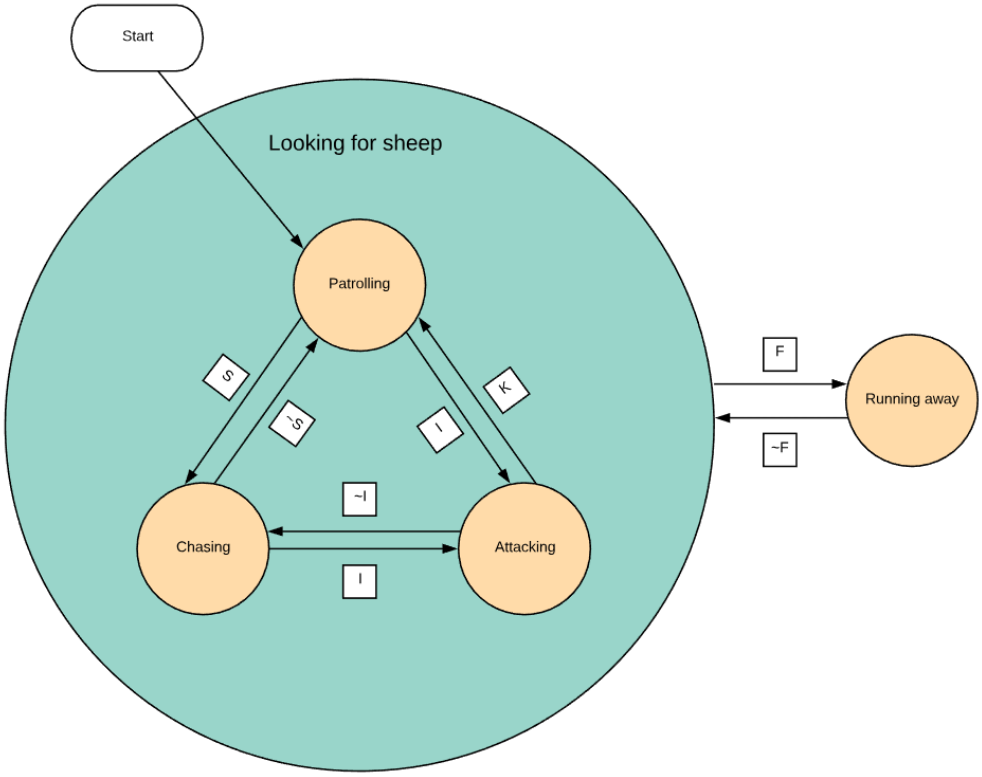
There are 6 **envents**(**transitions**) for the sheep FSM.

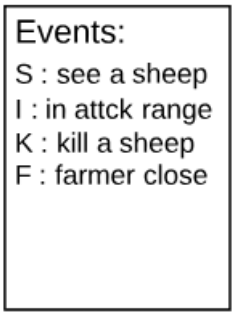
1. Alone: if(sheep can see other sheeps) false; else true;

2. Farmer close: if(distance between sheep and farmer <=10) true; else false;
3. Farmer closer: if(distance between sheep and farmer <=5) true; else false;
4. Wolf close: if(distance between sheep and wolf <=10) true, Wolf far away false;
5. Dead: if(sheep's HP values <=0) true; else false;
6. Wolf far away: if(distance between sheep and wolf >=15) true, Wolf close false;

For all events, we use Boolean variable to represent them.

- For the wolf:





We have 4 states for the wolf.

1. Patrolling: The wolf will walk around the way we give, there will be chance to find a sheep on the way.
2. Chasing: After the wolf find a sheep, we using pathfinding to let the wolf close to the sheep.
3. Attacking: If the distance between wolf and sheep is equal or less than 2, the wolf will attack the target sheep, 0.5 sec with 20% HP lose for sheep.
4. Running away: If the farmer close to the wolf(distance <=5), the wolf will running away, using flee farmer function.

There are 4 **envents**(**transitions**) for the sheep FSM.

1. See a sheep: if(distance between wolf and sheep<=20) true; else false;
2. In attack range: if(distance between wolf and sheep<=2) true; else false;
3. Kill a sheep: if(sheep's HP value <=0) true; else false;
4. Famer close: if(distance between wolf and farmer<=5) true; else false;

2.1.2 The algorithm for the FSM(HFSM):

Because we have lots of states and events(transitions) in both sheep FSM and wolf FSM, if we use the simple FSM algorithm to achieve, it will be a lot mess. For example, in sheep FSM, "random walking", "flocking", "flee farmer + random walking", "flee farmer + flocking" , " flee farmer", these 5 states can all have transitions with " Idle", if we using simple FSM, it will have lots of transitions between each other and it will also make lots of repetitive codes.

Finally we use the variation of FSM ---HFSM(hierarchy finite state machine) to achieve our diagram. HFSM allows us to only care about the transitions between the same levels, we don't need to care about the transitions inside others levels.

To achieve HFSM, we using "switch" statement to build code.

Using the sheep HFSM for example:

We have four levels to divide all states.

For base-level, there are only 2 states: "Alive" and "Dead"

For child-level-I, there are 2 states: "Flee wolf" and "Move"

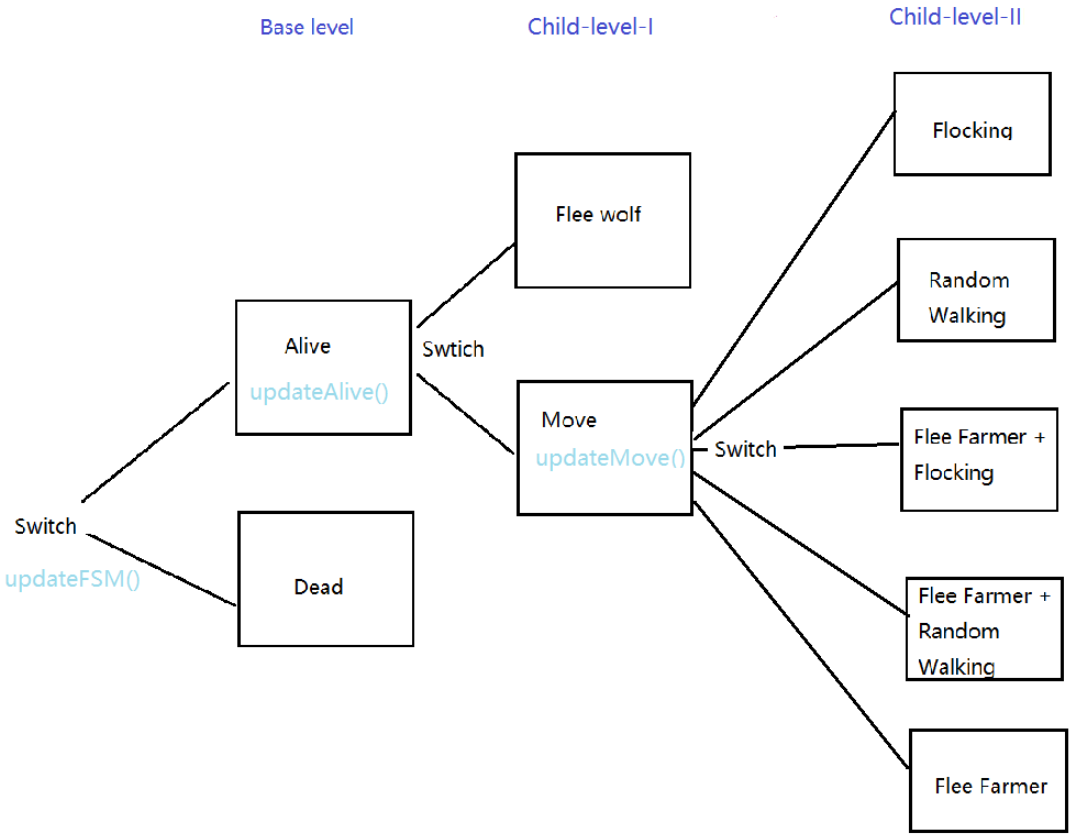
For child-level-II, there are 5 states: "Flocking", "RandomWalking", "FleeFarmer + Flocking", "FleeFarmer + RandomWalking", and "Flee Farmer".

We using enum to define all levels states.

```
5 references
public enum FSMState                                    //the base states
{
    Alive,
    Dead,
}

6 references
public enum FSMState_I                                  //first child states
{
    Move,
    FleeWolf,
}

19 references
public enum FSMState_II                                 //second child states
{
    RandomWalking,
    Flocking,
    FleeFarmer_RandomWalking,
    FleeFarmer_Flocking,
    FleeFarmer,
}
```

For each level of states, we using one switch statement, from top level to low level. And we write each switch statement in four different functions.

- The base level function is updateFSM().
- The child-level-I function is updateAlive().
- The child-level-II function is updateMove().



We only need to call the root level function “updateFSM()” in parent level, just showing in the above diagram. After switch to child nodes, it will automatically call the child level functions.

We using struct to define all transitions.

```
1 reference
public struct Transitions                                    //6 transitions
{
    public bool alone;
    public bool farmer_close;
    public bool farmer_closer;
    public bool dead;
    public bool wolf_close;
    public bool wolf_far;
}
```

And we using a function to update all transitions.

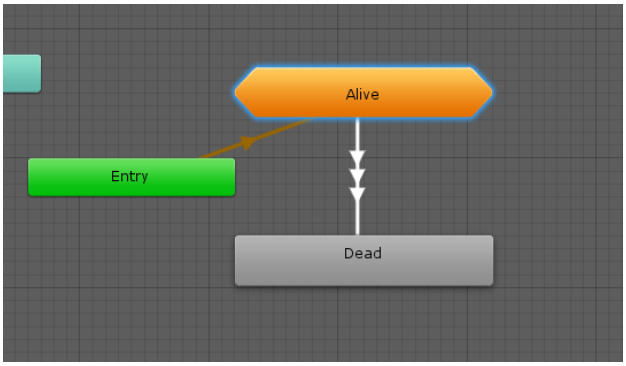
```
public void updateTransitions()                                    //update transitions
{
    if (flockingScript.getWolfDistance() <= flocking.MIN_DISTANCE_TO_WOLF)
    {
        transitions.wolf_close = true;
        transitions.wolf_far = false;
    }

    if (flockingScript.getWolfDistance() >= flocking.MAX_DISTANCE_TO_WOLF)
    {
        transitions.wolf_close = false;
        transitions.wolf_far = true;
    }

    if (flockingScript.numberOfSheepWeCanSee > 0)
        transitions.alone = false;
    else
        transitions.alone = true;

    if (slider.value <= 0)
        transitions.dead = true;
    else
        transitions.dead = false;

    if (flockingScript.distanceToShepherdV <= flocking.DISPERSE_DISTANCE_TO_SHEPHERD + flocking.MIN_DISTANCE_TO_SHEPHERD)
        transitions.farmer_close = true;
    else
        transitions.farmer_close = false;

    if (flockingScript.distanceToShepherdV <= flocking.MIN_DISTANCE_TO_SHEPHERD)
        transitions.farmer_closer = true;
    else
        transitions.farmer_closer = false;
```

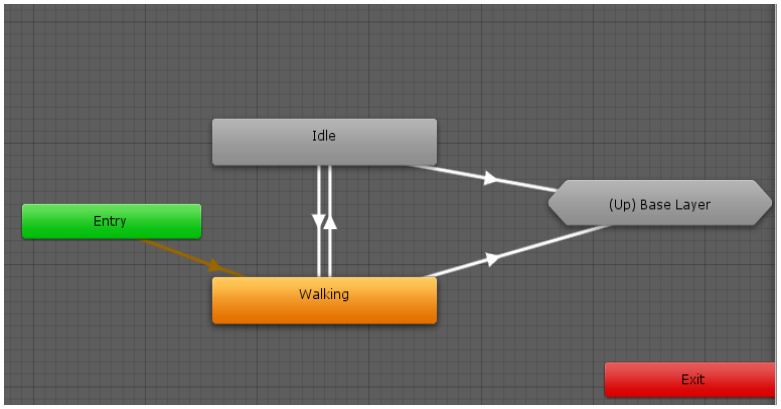
And we using this function determine all transitions true or false.

Finally, we just need to call “updateTransitions()” first and “updateFSM()” second in the “update()” function. It will update all transitions first, then update all states base on the updated transitions each frame. And the algorithm of wolf-HFSM is same as the sheep-HFSM.

2.1.3 Things are specific to our projects:

- Animation





For the animation part of sheep, we also using HFSM to build, in unity3d, it calls sub-state machine. Although the sheep has lots of different states, but there are only 3 different animations, for most of states, they are using the same animation- walking.

- Animation Speed

Although we are using the “walking” animation for most of states, we want to differentiate the different states. So we bring in the animation speed.

```
6 references
void setSheepSpeed(float speed)
{
    if (slider.value > 50)
    {
        sheepSpeed = speed;
        transform.GetComponent<Animator>().speed = speed;
    }
    else
    {
        sheepSpeed = LowHP_SPEED;
        transform.GetComponent<Animator>().speed = LowHP_SPEED;
    }
}
```

In both wolf and sheep FSM scripts, we have a same function to set the speed for both gameobject’s actually moving speed and gameobject’s animation speed. For example, when sheep random walking, it will have a low moving speed, accordingly, the animation speed will be a low level. But if a sheep flees a wolf, it will have a high moving speed, accordingly, the animation speed will be a high level to make the animation more real.

We also add a setting, if the sheep’s HP value <=50, the sheep will stay in low speed whatever states it is in.

- Slider (HP bar) and Timer

We add two slider(UI) crossed on sheep to make player can always see the HP for sheep in 3D world.

For wolf attacking sheep, we use timer in unity to control the wolf's attack speed.

```
wolfAttackSpeed = 1 / attckTimer;
attckTimer -= Time.deltaTime;
if(attckTimer <= 0)
{
    slider.value -= 20;                    //sheep HP -20, full HP is 100.
    attckTimer = 0.5f;
}
```

When wolf in attacking state, it will update attacking() function each frame in update() as the above picture. So in each 0.5 second, the sheep will lose 20% HP value.

2.1.4 Surprise and problem:

- Surprise:

    1. I have another game project, I was making fight system using collision and trigger event to handle. But it's ok for melee(short-attack range) heroes, but it's difficult to achieve for long-attack range heroes(like archer). But after I used timer in this project, I think timer is a good way to build the fight system, timer is also a easy way to handle the attack speed. We don't even need collision after using timer to build fight system.

    2. In the same project I talked before, we are difficult to handle the states and animations in game, there are lots of mess, we only use "if else" to do all things. But after I study FSM/HFSM, I think I can use this knowledge in my game project, it is a really good way to handle states and avoid mess.

- Problem:

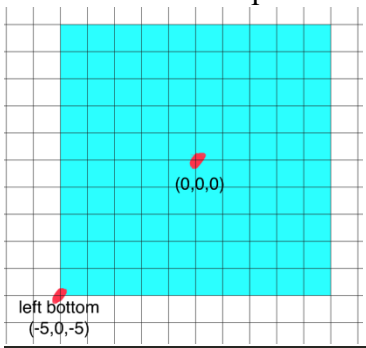
    HFSM is better than FSM, but it still has a little problem. When the hierarchy becomes lager, it will make the states in each hierarchy coupled tightly. And it hard to update new state and modify old state. If we don't want the hierarchy becomes lager, we have to make different hierarchy in same level. There may be some states

that are a re-write of state in other hierarchy but in same level. I find a better way to achieve the states is using Behaviour Tree. And I will try to use Behaviour Tree to replace the HFSM in the second assignment.

2.2 Path Finding (A*)

2.2.1 Node object: contains parameters of x value, y value, word position (Vector 3), Gcost, Hcost, Fcost and parent node.

2.2.2 Find left-bottom: Let the position centre of the ground to (0,0,0). The distance from the centre point to the left edge will be the half size of the ground X and the distance from the centre point to the right edge will be the half size of the ground Y. So move to the left edge then move to the right edge then we get the left-bottom point.



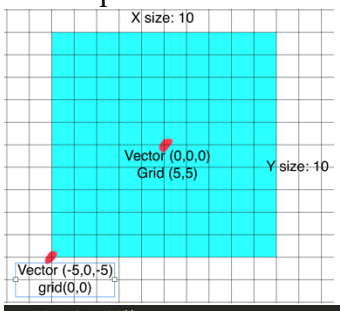

```
ground = GameObject.Find("Ground");
GroundSizeX = ground.GetComponent<Transform>().localScale.x * 10;
GroundSizeY = ground.GetComponent<Transform>().localScale.z * 10;
//Ground size devides node size to get the grid size
GridSizeX = Mathf.RoundToInt(GroundSizeX / nodesize);
GridSizeY = Mathf.RoundToInt(GroundSizeY / nodesize);
//When the grid start from (0,0), the start posision of the ground should be the left-bottom-most position
//To calculate left-bottom position, find the center position of the ground first,
//then move left for have size of the ground.x, and move down for half size of the ground.y
LeftBottom = ground.GetComponent<Transform>().position - Vector3.right * (GroundSizeX / 2) - Vector3.forward * (GroundSizeY / 2);
//After calculate grid size, initialize grid
InitGrid();
```

2.2.3 Create grid: Determine a size for node (default as 1). Then get the X size and Y size of the ground in the world (a plane object). Divide the ground to small cells by the size of node then the cells will be the nodes to generate the grid.

2.2.4 Initialise grid: I choose to use 2D array to store the nodes in the grid and generate the grid. The start Node of the grid (0,0) will be the left-bottom point of the ground (calculated in step 1). The size of the inner array is the Y size of the grid and the outer size of array is the X size of the grid. Each node in the grid will be related to a world position and the world position (a Vector 3) will be recorded as a parameter in each Node.

2.2.5 Find the position in the world from a Node in the grid: The x value of the Node times node size representing how far the world position moved right from left-bottom position and the y value of the Node times node size

representing how far the world position moved forward from left-bottom position.




```
void InitGrid()
{
    //Using 2d array to implement grid
    grid = new Node[GridSizeX, GridSizeY];
    //The autual position on the ground will moving with the increasing of both x and y of the grid
    for (int i = 0; i < GridSizeX; i++)
    {
        for (int j = 0; j < GridSizeY; j++)
        {
            Vector3 ActuralPosition = LeftBottom + Vector3.right * (i * nodesize) + Vector3.forward * (j * nodesize);
            bool isobstacle = Physics.CheckSphere(ActuralPosition, nodesize / 2, Obstacle);
            Node cell = new Node(i, j,ActuralPosition,isobstacle);
            grid[i, j] = cell;
        }
    }
    gridexists = true;
}
```

2.2.6 Representing obstacles in grid: The obstacles in the game are cube objects with layer named “Obstacle”. For every position in the world I will CheckSphere for this position, if CheckSphere is true then the corresponding node of this position in the grid will be set to obstacle.

2.2.7 Check target is inside unwalkable area or not, if outside unwalkable area then start pathfinding algorithm. If target is inside unwalkable area then do nothing because there will be no path.

2.2.8 After initialize grid and nodes, start A* algorithm:

(1) Get start (Vector 3) and target (Vector 3) and translate them to nodes in the grid: calculate how far the vector from left-bottom position on both x direction and z direction, then let these distances divided by nodesize and get round of the result to calculate the x value and y value of the node. Then using the x value and y value to get the Node from the grid initialised in step 4.

```
//Method to convert ground positoin to grid position
public Node WorldToNode(Vector3 world)
{
    float VectorXMove, VectorYMove;
    VectorXMove = world.x - LeftBottom.x;
    VectorYMove = world.z - LeftBottom.z;
    int GridX = Mathf.RoundToInt(VectorXMove/nodesize);
    int GridY= Mathf.RoundToInt(VectorYMove / nodesize);
    return grid[GridX, GridY];
}
```

(2) Create a List (“explored”) to store nodes that have been explored as previous nodes’ neighbours but haven’t been choose as the smallest neighbour. Create a HashSet (“pathlist”) to store nodes that have been choose as the smallest neighbours of previous nodes. Set Hcost, Fcost and Gcost of the startnode to 0 and add startnode to “explored”.

(3) While the size of “explored” is greater than 0, repeating the following steps:

Find the nodes with smallest Fcost amoung all nodes in “explored” list. If there are more than one nodes that has the smallest Fcost, then compare the Hcost and select the node with smallest Hcost. After find the smallest neighbour node, set this node as “current” node, add this node to “pathlist” and remove from “explored” list.

(1) If “current” node is Endnode, then get out of the loop that means the path has been found, retrace from Endnode to get the path.

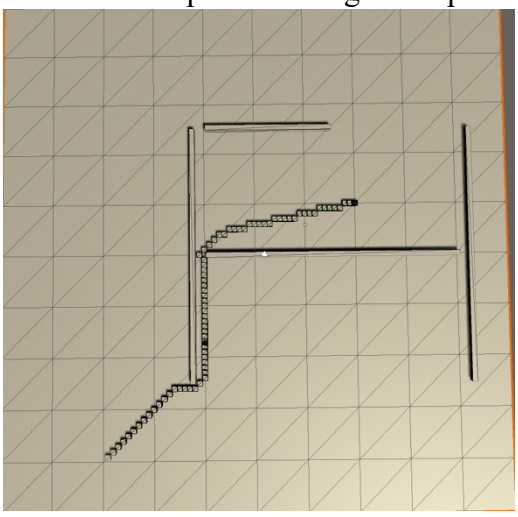
(2) Get the neighbour nodes of “current” node. Iterate the neighbour nodes: If neighbour node is an obstacle than do nothing and move to next neighbour node. If neighbour bode is not an obstacle:
[1] Get the Gcost of current node and let the value plus the distance between current node and this neighbour node as a new Gcost.
[2] Calculate Hcost for this neighbour node: Calculate the distance between this neighbour node to endnode *[1]:

```
float GetDistance(Node nodeA, Node nodeB)
{
    Vector3 vescost = nodeA.getWorldPosition() - nodeB.getWorldPosition();
    return vescost.magnitude;
}
```

[3] If this neighbour exists in the “pathlist and new Gcost is greater than Gcost stored in the neighour node, discard this neighbour node and move to next neighbour node. If this neighbour doesn’t exist in the “pathlist”, go to next step.
[4] If this neighbour doesn’t exist in the “explored” list or new Gcost is smaller than Gcost stored in the neighour node, go to next step. Otherwise discard this neighbour node and move to next neighbour node.
[5] Set new Gcost and Hcost to this neighbour node and add these values together to get the Fcost and store Fcose in this node.
[6] Set this neighbour node’s parent to “current” node and add this neighbour node to “explored” list.

2.2.9 Create a list of node “finalpath”, set Endnode as “temp” node, while temp is not the Startnode, keeping repeating following steps: add “temp” node to final path, get “temp” parent node and set it to “temp”.

2.2.10 Reverse “finalpath” list to get real path. Finish A* algorithm.



- Any surprises you or problems you had:
  After the A* algorithm find a path, I gave the path to the game object to let the object to move alone this path, however the game object doesn't follow this path and sometimes the object stacks in the obstacles although the obstacles are not included in the path. But after I read the book *Unity 4.x Game AI Programming* I realized to solve this problem I need to introduce steering behaviours to pathfinding.

- Choices you made that are specific to your project:
  Before actually run the A* algorithm, I will check the position of the start point and end point, if any one of them inside unwalkable area then there will be no path and the it won't go to the algorithm part which could save resources.

- Any deviations from textbook algorithms you made, and why:
  For the open list I should use SortedSet because the HashSet will has better performance than list. But I tried sortedset but there were some unexpected bugs so for now I will use list and in later stage I will try to use sortedset for close list.

- References to any other works that you may have used or built upon:
  [1] Kyaw, Aung Sithu, Peters, Clifford., and Swe, Thet Naing. *Unity 4.x Game AI Programming*. Birmingham, UK: Packt Pub., 2013.

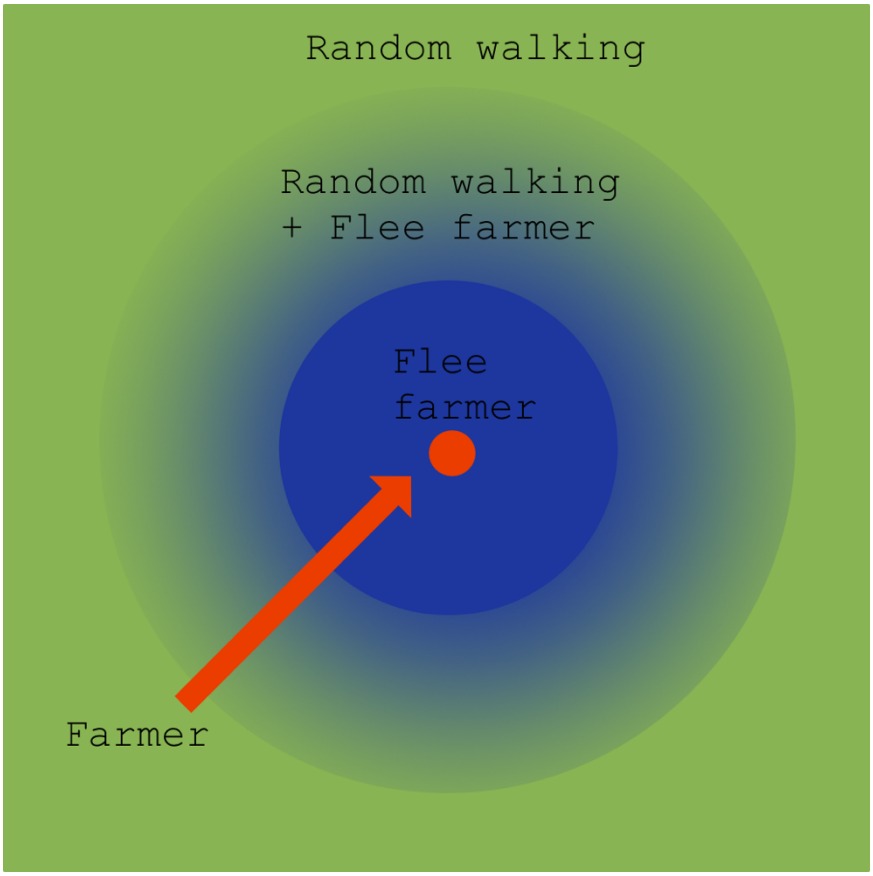
3 Flocking

- How the algorithm works:
  - Each sheep has a target, this target is the desired position for the sheep, and is build-up of multiple components depending on which state of the sheep FSM we are in:
    - RANDOM_WALKING
      - Here the target I chosen a random and follow for 2 seconds before a break of 4.5 to 9 seconds of no movement, before a new target is selected and the process repeats.
    - FLOCKING
      - Here we follow the normal flocking behaviour, I.e. our target is a weighted sum of the independent target from separation, cohesion and alignment.
        There is also a l little random walking when flocking, to make the flocking less predictable, and to make it look like the sheep move from spot to spot and grass.
    - FLEE_FARMER

- Here the target is just in the opposite direction of where the farmer is, in relationship to the sheep.

- RANDOM_WALKING_AND_FLEE_FARMER

  - This is a combination of what we do in random walking and flee farmer, du the combination is weighed, depending on how far away the farmer is. This is done to avoid state flickering where a sheep switches between fleeing the farmer and random walking, each frame. The gradient weighing is illustrated below, where green is the weight put on random walking, and blue is the weight put on fleeing the farmer:



- FLOCKING_AND_FLEE_FARMER

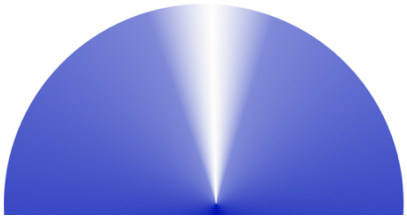
  - Flocking and fleeing the farmer is based on the same concept as described above, but with flocking instead of random walking.

- FLEE_WOLF

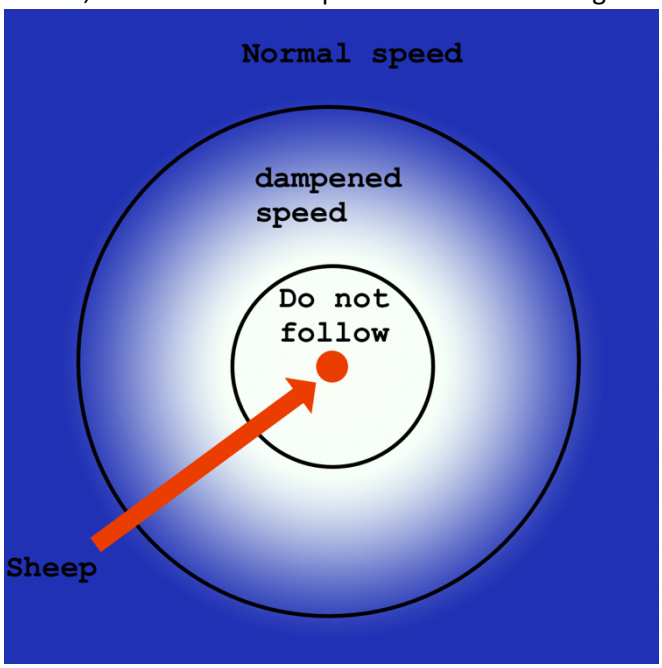
  - When we flee from the wolf the target set in the opposite direction of the wolf. When we are in this state the sheep can also run faster than normal, to simulate normal chasing behaviour. Since we do not have and gradient in behaviour as in the picture above, the distance to the wolf before we stop fleeing is larger than the distance we start fleeing. This

combined with that fact that the wolf is always moving, is enough to avoid state oscillation in practise.

        - DEAD
            - Here the target is where the sheep lies.
    - After the target is set the sheep needs to approach the target, this is done by a combination of angular velocity and regular velocity:
        - **Angular velocity:** The sheep always try to face the target, which means the sheep will turn to. To make this turning look natural we use dampening when we get close to the desired direction. This is illustrated below, where the blue colour represents the speed of rotation:



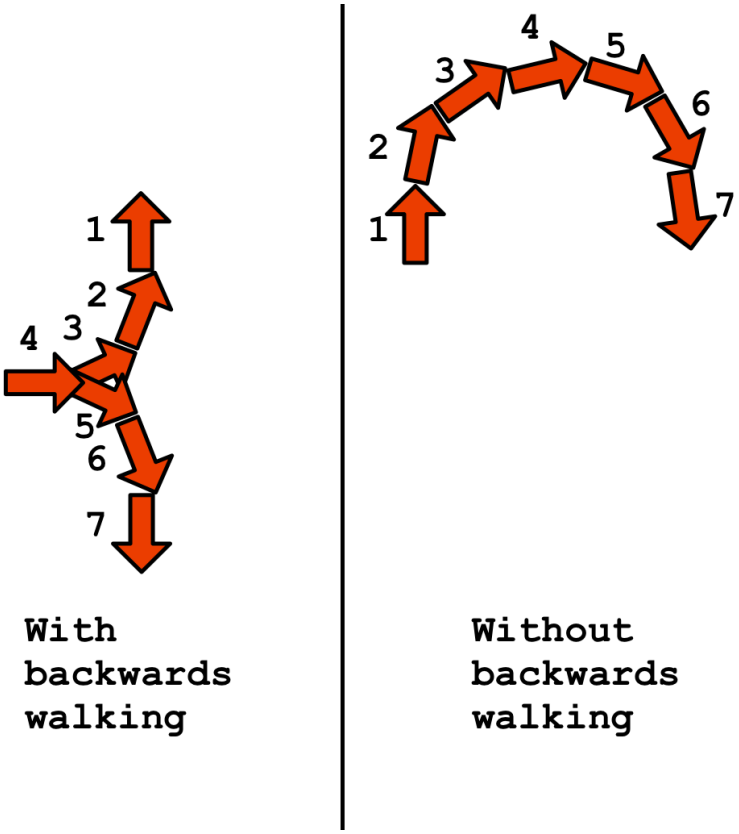
        - **Velocity**: We always move the sheep in the direction it is facing, so no sideways walking. As with angular velocity we also have dampening with velocity. If the target is within 2.5 meters of the sheep, the sheep won't even bother to try to get to it, if it is within 3.5 meters the sheep will go towards to target but not with full speed. Only of the target is more than 3.5 meters from the sheep will the sheep go towards the target with normal speed. The lack of moment when the target is close, is to prevent the sheep from flickering since the target always moves a little bit. This is illustrated bellow, where blue is the speed we follow the target with:



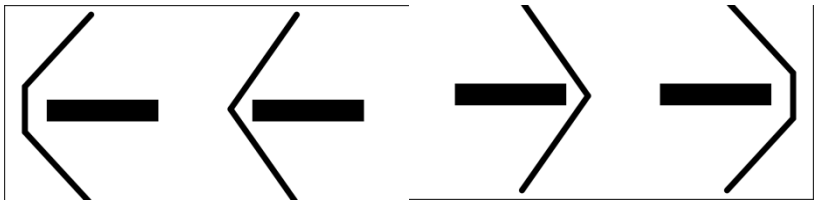
- Things that are specific to our project
    - **Random walking**: At first, we implemented random wandering, as described in the textbook. But for sheep to just wander around randomly looked wrong. Therefore, we tried what we have now which is “walk in a random direction for 2 seconds and

wait for 4.5 to 9 seconds", this look way more natural for sheep. It also mimics the way a sheep eats the grass where it is, before moving to some fresh grass.

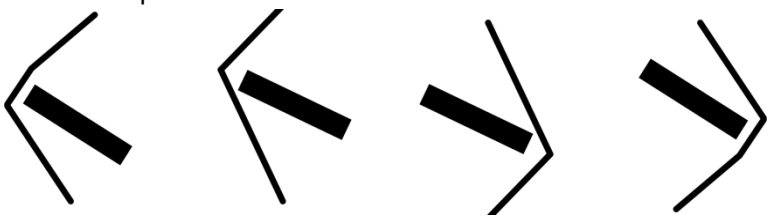
- **Walking backwards**: Sheep sometimes hit obstacles when turning, because they had a to big turning circle. To solve this we added so sheep can walk backwards, still following the direction of the sheep. With this a sheep can now "back away" like a car on much less area. This is illustrated below:



- **Object avoidance**: First we added A* path finding to our sheep, but this look to unrealistic, so instead we added only simple object avoidance.
  This object avoidance simply works by:
  - Do I have line of sight to my target, if yes go there.
  - If not find the object blocking, and try the 4 basic paths around, and take the shortest one that works.
    Here are the 4 basic paths illustrated, these are made so we can easily path find around corners and fences.



These paths depend on the object blocking your way, ex. if the fence is rotated the paths will follow:



The sheep do not follow the hole path, it only uses it to navigate around an object. Most often when the sheep come to one end of the fence, it will keep going away from the fence, instead of wrapping around, since the target moves constantly.

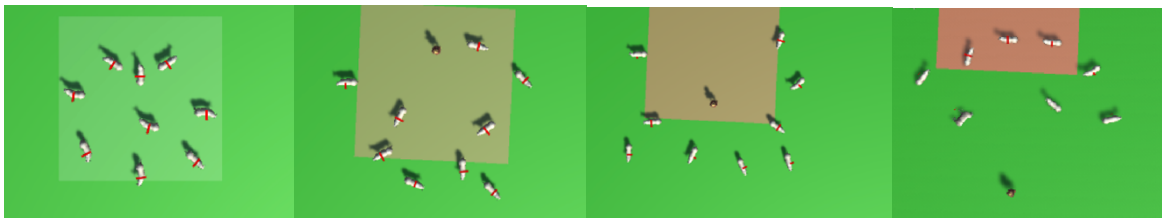
This is illustrated below, where we in the first image can see sheep that are stuck with no way out. The second image shows the left barrier removed, and therefore the sheep can now escape:

    - If none of them work, just go towards the object anyway. (so, they do not freeze) The reason we only added the 4 simple paths around objects was more complex paths and A* path finding seamed unrealistic for a sheep for follow, and indeed looks odd when used.
- Any deviations from the textbook algorithm you made
    - **Lack of cohesion**: We put a low weight on cohesion, since sheep do not just wander as a flock, most of the time they stand still and eat grass. So, a higher weight on cohesion looks odd for the sheep.

Two examples of flocking are shown below:

First, we have an example of two groups of sheep that see each other and join to form one big group.

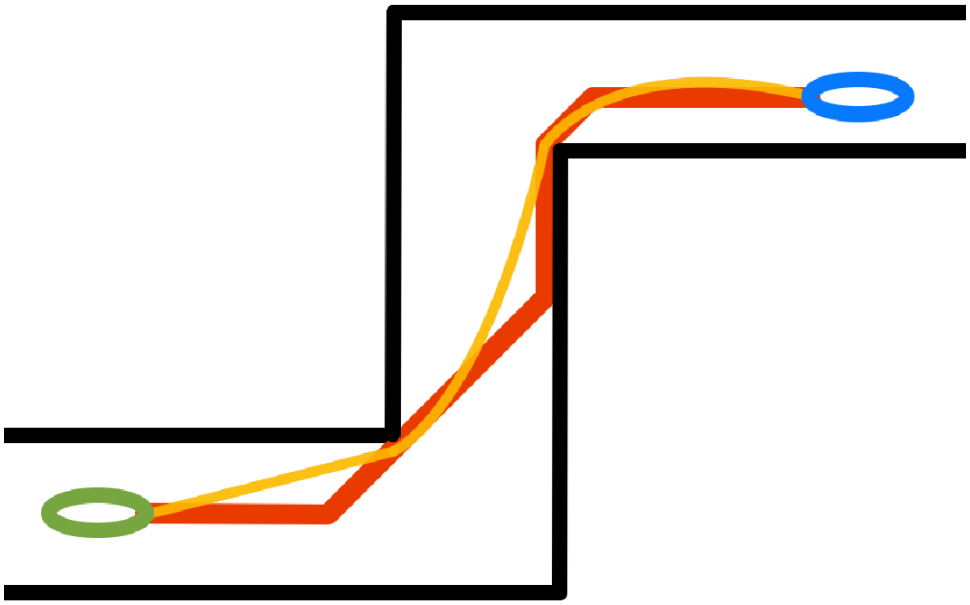
Second, we have a group of sheep that splits for the farmer and then re-join after he is past:



Wolf steering

Since the A* path finding used on the wolf is grid based, the path it finds do have some sharp corners. To smooth this out when moving the wolf, we apply a force to the wolf in the direction of the grid point furthest along the path, that we have a line of sight to. To make lower the number of times the wolf is stuck on corners, we use 3 rays: one directly from the wolf to the grid point, and 2 more parallel to the first offset perpendicularly, to make sure the path we are following can is wide enough for the wolf.

The smoother path is illustrated in the image below. Here the wolf is the green oval, the target is the blue oval, the red path is the path found by our A* start part finding, and the yellow path is the path the wolf ends up walking. This steering is also nicely illustrated in the scene "Scenes/PathFindingExample" where the wolf finds its way out of a maze.



Contribution

All members have written about what they implemented, which is shown in the table below:

Person	Contribution
Haixiao Dai (s3678322)	Path Finding for Wolf: Algorithm, Testing, Grid generation for A*
Yifan (s3672150)	FSM Algorithm for both sheep and wolf, Testing Model
Mads (s3799147)	Flocking and Steering behaviours for sheep, Steering for wolf, Fences generation, Testing