The AI Development of “Nazo”

Game Overview

We are Team 9 and the game that we have decided to create for this class is a PC farming monster wave fighting game called “Nazo” made in Unreal Engine. Though farming games are fun and relaxing, they often lack progression. We initially wanted to implement a puzzle section in order to add a story to the game and give the player a feeling that time is moving and that there is indeed an end that they can achieve. However, since the semester is short, we did not have time to go with the puzzle idea and decided that it was best to make a farming game that had monster waves for the player to fight to progress through a story.

The story of our game follows a farmer (the player) who inherits a farm from her late grandfather, however the player is in debt and needs to find a way to make money in order to save herself and her farm. In the day cycle of our game, the farmer focuses on growing her crops to sell for profit. The money earned can be used to buy more crops to grow for profit to pay off the farmer’s debt, or to buy fences in an attempt to help protect the crops from being destroyed in the night. When night falls in our game, beetles will start to come out in waves and try to eat the farmer’s crops. The amount of beetles in each wave will increase as the game progresses. To defeat these waves, the player must utilize their slingshot to shoot at the beetles, build defenses, and defeat the beetles.

AI Development - FSM

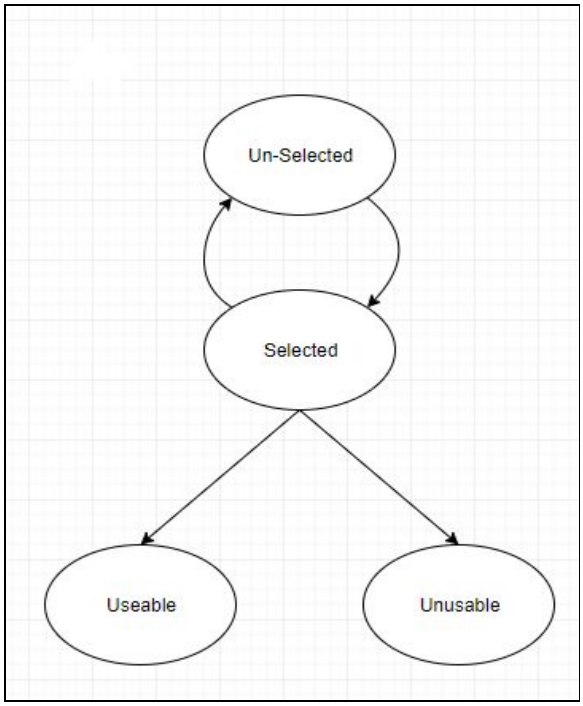
1. Crops

One of the first simple AI systems we implemented were the crops. They move from state to state based on the condition of the soil under them and the number of days that they have been watered. A seed can only be planted if the soil is tilled, and will only grow if it is watered. We also used good OOP practices by making a parent seed behaviour type that other crop types inherit from, so that they all have various growing states and behaviours. Each plant will have a total number of days it needs to grow, and the number of watered days needed for each stage is based off of that using very simple math. If the plant is watered once it grows to a sprout, 1/3 of the way it

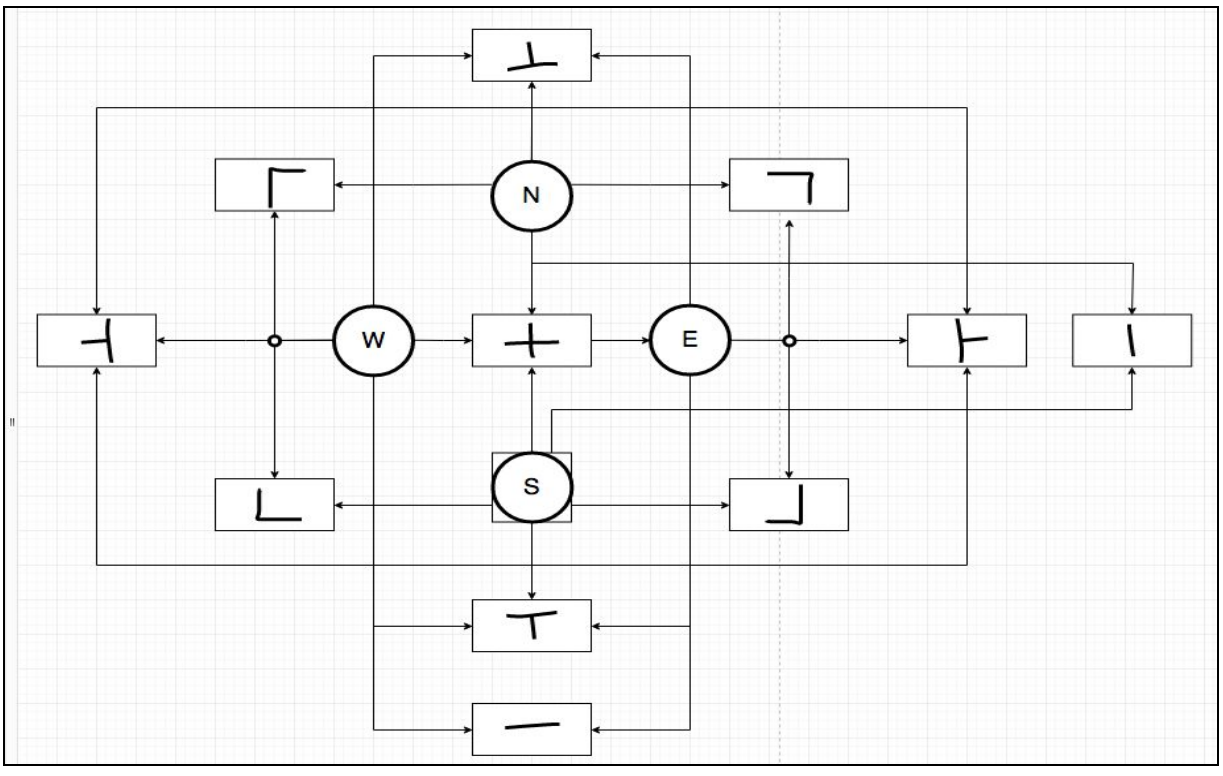
becomes the plant, and on the final day it can be harvested, and will increase the number of objects in the player's inventory.

2. Grid System

Since each plant needed its own planting square, it became apparent that we needed some way for each tile to act differently than the other, and also show the player what they were doing, we decided to use a grid. Once the game begins, the grid spawns from a construction script, and uses public variables for what size it needs to be. It then checks if the camera's forward vector is hitting it or not. If it is being hit, then that piece becomes the selected grid, and the previous piece will be deselected. Since it is consistently checking if the player is looking at each grid piece, we decided to make a grid tick that would tick much more slowly. That way, it wouldn't be checking every framerate and costing processing power. Then it checks through collision if the square under it is valid for whatever task the user wants to do. If the object type isn't soil, it will block the user from planting a seed, and if the object type is soil, it will not let the player place defenses.



3. Fences



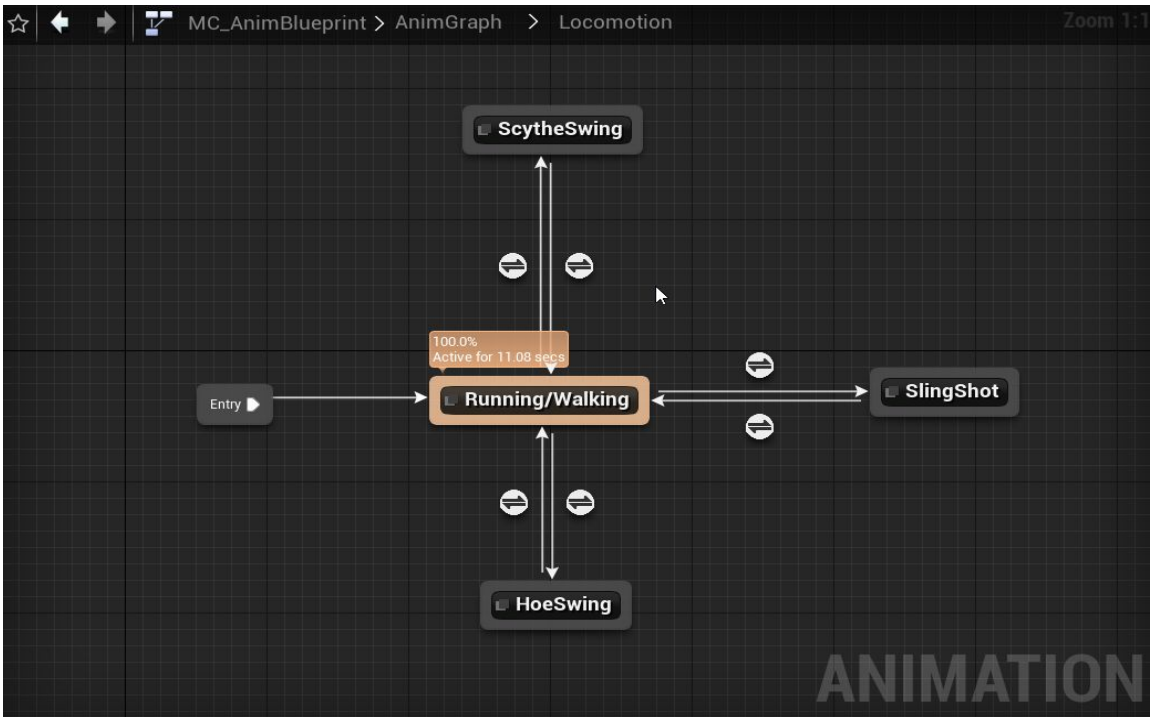
Since we wanted the fences to lock into each other to form cohesive barriers, and not have gaps in between, we decided the fence needed to be able to detect if another fence type was next to it, and then assume the proper mesh. We placed 4 collision boxes for each cardinal direction: North, South, East, West. It then detects which boxes are being overlapped with a fence object type. Depending on how many are being overlapped, it will change the mesh to one of four types. It will then rotate the mesh so that the inner mesh matches the number of coordinates being overlapped.

4. Weather

Another part of our game that had AI is the weather system. Every 30 seconds that passes in game has a 10% chance of having rain. When it rains, the sky would turn dark. The sky will return to its natural state when the rain stops based on a timer. We are currently still implementing having the soil become watered when the sky rains through an interface, but will offer the player some relief from watering each plot of land over and over again, thus breaking up some of the tedium that can arise. The system alternates from no rain to rain state.

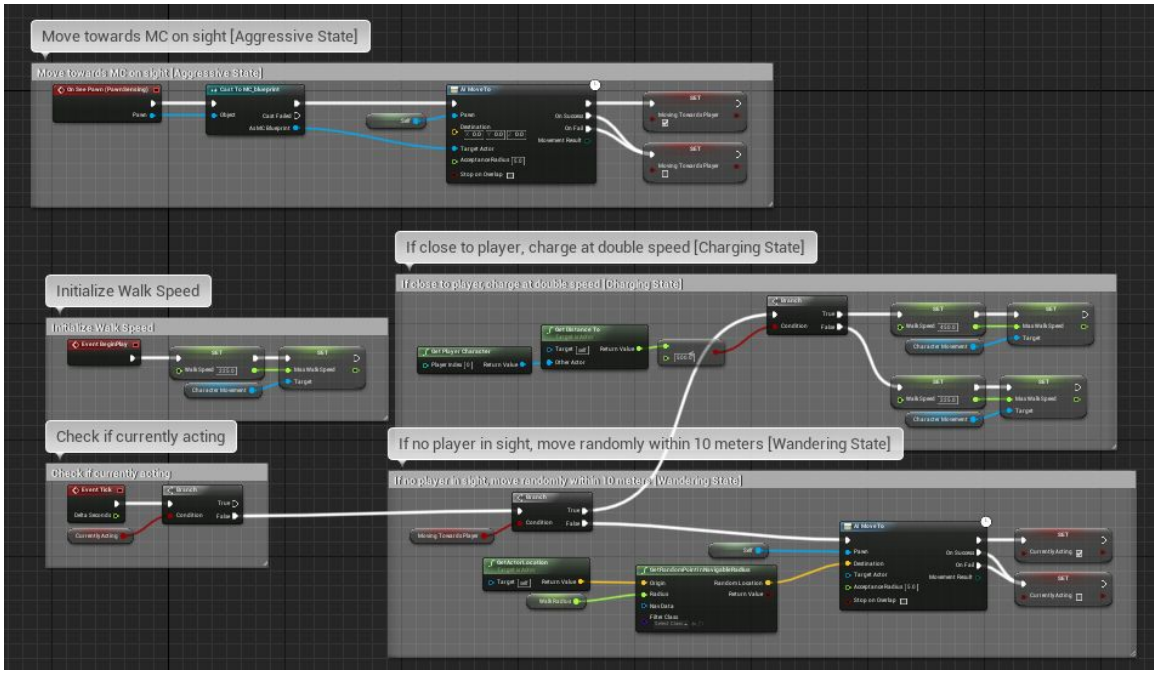
5. Animation/AnimGraph

As we started to add on animations for various tools and weapons, it became obvious that having the animation of running stop, but the character continuing to move would become jarring to the player. We decided to move to a state machine based animGraph that grabbed various boolean values from whether or not a player was doing an action, and then playing that action above a certain bone, and the running/walking blend state on the bottom. It would stay in the state until the animation was done, and the boolean became false.



6. Beetle - Original AI

The original AI for the beetle enemy was based on a finite state machine. Each frame it would check a flag boolean variable as to whether it was currently acting and if not then it would assign a new task to the beetle. While this was acceptable for our original design (mainly because the beetles were never intended to do anything more than attack the player on sight), this didn't allow for performing more complex tasks such as 'moving towards the farm after a random number of wanders' without drastically expanding upon the blueprint in a way that made the entire design become overly complicated.



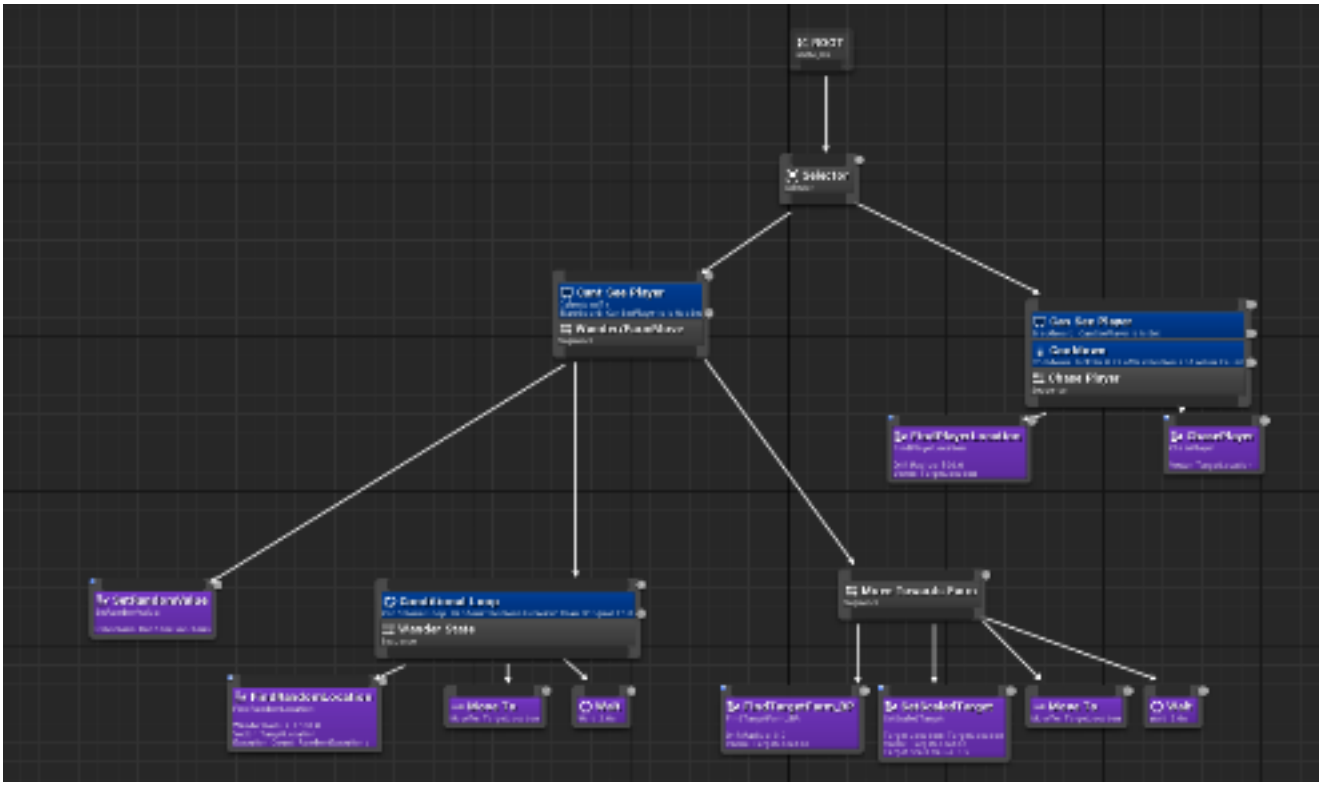
The Decision to move to Behavior Trees

While the aesthetics and readability of the code was important to us, the main thing that drove us to changing the AI from a FSM to a Behavior Tree was that we could separate the AI from the Beetle Blueprint. This allowed us to modify tasks within the behavior of the beetles without affecting *all* of the beetles. For example, should we need a beetle to behave differently in the tutorial then we could just spawn a generic Beetle_BP and assign an AI controller that utilizes a different tree instead of having to create a duplicate of the Beetle_BP that had the modified behavior hard coded into it. While we wound up not needing to utilize different behaviors for different beetles, this change did allow us to create a child based on the beetle behavior tree and derive the AI patterns for a new wasp enemy.

AI Development - Behaviour Trees (BT)

1. Enemy Beetle AI

After the transition from the FSM to the BT, the beetle managed to retain all of the states that were initially part of the FSM. The beetle still wanders, chases, and charges as previously but we've added a random variable on how many wander cycles it completes before performing the new action of moving towards the farm. In order to accomplish this, the beetle searches the map for a FarmTarget actor and calculates the vector needed to move towards it. It does this after 0-3 random wander cycles so that the beetles don't all move uniformly. When the crops are in sight, the beetles track onto them the same way they do a player character and damage the crops if left unchecked to destroy them.



We decided to use some strategies we saw in class to smooth out the movement. Instead of finding and following a random vector, the beetle seeks a trigger box that changes to restricted spots within a radius around the beetle, and always keeps a certain distance from it. This smooths out the movement since it's not just going to and then arriving to a random spot.

2. Shopkeeper NPC

Since finding some success in how well the beetle AI scaled with the BT, we decided to also make our friendly shopkeeper NPC use a BT as well, just in case it needed to scale, and since we also wanted to use an AI controller for the way it moved. The NPC also needs to follow some pathing, but instead of implementing a spline based movement system, we also used the trigger boxes to move to certain locations based on time. Should we want to also have the NPC possibly attack the beetles or fences in it's way in the future, it will be easier to add that behaviour.

Iteration

After our game testing day where other students were invited to play our games and review them so we can fix our games for the better. We found a list of things that needed to be fix for the final showing of our game, such as the beetle's movement.

During testing, we found that the beetle's AI code was working well, but it was best to smooth out its movement since they were said to be moving too robotically. Another problem that came with the beetles was that their spawning was not working properly. The code is now fixed so that the waves will spawn properly.

After all the fixes from game testing day were made, we have done another iteration of the game to see what features were working well and what features needed to be fixed. In general, the beetle movements were well received since they were more smooth and they have clearer attack animations. The beetle waves were spawning properly but players found a point where the waves got too powerful, so the amount of beetles spawning had to be adjusted accordingly.

Some other fixes that we had were related to making the mechanics of the game easier to understand. This iteration showed that even though we still need to make a few more changes to make, many of the fixes to the game were successful and players enjoyed the game more.