AI Strategy Design Document

Summary:

The **GoapAgent** uses GOAP (Goal Oriented Action Planning) to decision make based on the different world states and the different prerequisites inside each action. The **GoapAgent** uses individual GOAP actions that are **MonoBehaviours** if those actions are attached to the **GameObject.**  The agent also stores other various useful items like *currentPlan* and *currentAction* variables which contribute to executing the plan. To create a plan however there must be a goal set inside the *goals* Hash table which is then used later in the **MakePlan** method to create a plan using the **GoapPlanner** class. The **GoapAgent** also has references to the combatant, recipe and other things for the actions to be able to access.

The **GoapAgent** has some setup methods that help the AI know what is available, in **Awake** the *combatant* is grabbed from the **GameObject** and the *agent* is set in the *combatant.* In **Start** the **ResetWorldStates** and **Initialize** methods are called which set the available actions and the world states to what they are meant to be on start.

Inside of the Update loop the *combatant* is made sure not to be *dead* or damaged and that the game has not ended then if there is a plan it progresses the plan at the state it is currently in using **ExecutePlan**. but if there is not a plan it will call **MakePlan.**

Actions:

The actions are what determines what the AI does using the *combatant* to perform the actions over a certain amount of time if applicable. Each action has its own requirements to and a reaction to be performed. If those requirements are not met it will not succeed inside the **GoapPlanner** and they will not be put into the plan.

Most actions when completed set the world state to what they have accomplished and then return true otherwise they return false and are still queued to run in the next update. Actions can exit early however if the requirements for that action are no longer met which will stop the plan entirely and force the AI to change its state and pick a new plan.

Plan:

In order to be able to make a plan the AI must have a goal, or it will return null and in order for the plan to execute there must be an action plan. What sets the goals however if the *stateMachine*, the *stateMachine* will use the current state to call a method within the **GoapAgent**class that sets the corresponding goals and world states if applicable. Setting a new goal however will also call the **ForceQuitPlanMethod.**

AI Strategy Design Document

The **ForceQuitPlanMethod**, when called will set *exit Plan* to true and make sure that the AI is no longer aiming. There are three other methods within plan **CheckForPlan, MakePlan** and **ExecutePlan.**

* **CheckForPlan** performs the combatants **StopAIMovement,** takes the current goals world states an action and returns whether a plan could be formed from that using the **GoapPlanner.**
* **MakePlan** creates a plan the same way as **CheckForPlan** does except it checks to make sure the AI doesn’t get stuck trying to attack repeatedly without success.
* **ExecutePlan** check to make sure there is a plan before executing. If there is a plan and the plan is not meant to *exitPlan* than it will perform the *currentAction* which if the first in the queue of *currentPlan* until that action is done in which then the *currentAction* will change and the execution will continue until the plan has reached its end.

Black Board:

The **GoapBlackBoard** holds the *WorldState*s which is shared universally throughout the scene, these states are used to check the requirements of the actions, so they can be performed. The **LogState** method will add or replace a sate when it is called and a new *WorldState* is passed through. However, each of the AI have their own key for the world states to ensure there is no crossover between AI. While the actions use the world states to check whether they can accomplish a task they are also able to use the consequences of each action to replace the world states for better predictability of what will come next.

The **GameManager** also holds loads of useful data that the agent uses to make decisions and perform actions. The actions will use the various public variables from the **GameManager** like:

* *combatants*
* *ovens*
* *bombs*
* *gameState*
* grid
* *combatantsLeft*
* *recipes*

Difficulty:

In the future as difficulty increases so will the time to do actions and the chance to be able to do actions will be increased. They will be less accurate with the lesser difficulties and more evasive in the higher. They will prioritize the player slightly more as well.

Feedback:

The feedback I got from Joel was quite insightful. He wondered why I used **MonoBehaviours** for my actions instead of just creating them at runtime, I also had

that same question and will be changing that in the future. He was also confused about how the movement works which is valid as the movement is quite convoluted and could be shortened by a great amount. He suggested I change the actions which I will be doing in the future. Nates feedback was different however as he did not understand the fundamentals of how GOAP works he also wondered why I used it instead of masking a much simpler decision-making algorithm. The response was that I wanted a challenge, and I was too lazy to build a new one with the amount of time left. His suggestion was to make the AI wander more, so they don’t always come after you. I took his suggestion and implemented it but saw miniscule improvement so I am going to further think on what I might need to make the AI deviate from their path a little more.

Table Of Contents:

TODOs for each section:

* Assignment 2 Pathfinding algorithm

//TODO: For Rob AI pathfinding assignment 2

* GoapAgent regions for ease of readability

//TODO: For Rob Goap Agent

* Goap Blackboard

//TODO: For Rob GoapBlackBoard

* State Machine

//TODO: For Rob State machine

* Game Manger

//TODO: for Rob GameManager

* For actions see folder path: Assets/Runtime/Scripts/AI/Goap/Actions