Project Robot Jousting
Herkansing KGDEV3, opdracht 2
Frans Huntink
Game development

Inleiding

Voor opdracht 2 van de kernmodule game development 3 heb ik gekozen om mezelf uit te dagen. In plaats van een behaviour tree uit te werken, heb ik besloten om mijn eigen utility system uit te werken in Unity. In dit document leg ik aan de hand van screenshots en illustraties uit hoe dit in zijn werk gaat. Ik zie dit project niet alleen als een herkansing, maar ook als een case study voor mezelf. Als het utility systeem werkt zoals ik het beoog, is het een interessante aanpak voor eventuele toekomstige games.

Doelstelling

Ik heb mezelf voorgenomen om met dit systeem een vorm van *emergent behaviour* te creëren. Dit wil ik doen door een spelwereld met simpele regels te combineren met een agent die complexe besluiten afweegt op basis voorspellingen en gewichten.

Het gedrag van de agent moet makkelijk af te stellen zijn, en dit wil ik terugzien wanneer er twee of meerdere agents met elkaar de strijd aangaan.

Het systeem

Een utility systeem bekijkt heel simpel gezegd van welke actie de *agent* het “meest gelukkig” wordt op dat moment. Dit geluk druk ik uit in een percentage. De agent itereert over alle mogelijke acties (in mijn systeem genaamd *possibilities, zie kopje hieronder)* en voert vervolgens de actie uit met de hoogste utility waarde.

Possibilities

Iedere mogelijke actie die een agent kan ondernemen noemen we een *possibility*. Dit kan ik allerlei vormen komen, denk hierbij aan rennen, idlen, een item consumeren of aanvallen. In mijn framework implementeert iedere actie of bruikbare item het *IPossibilityTarget* interface. In dit interface wordt slechts een variabele vereist, namelijk het *type* van de possibility. Dit is een enumerator met de volgende entries:

- **APPLY_ITEM**
- **NAVIGATE**
- **ATTACK**

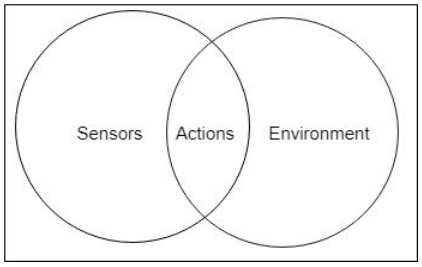
Senses

Om de possibilities in kaart te brengen voor een agent, maak iedere agent gebruik van de *AgentSenses* class. Deze class checkt welke objecten er zich in de naaste omgeving van een agent begeven. Ieder object dat het *IPossibilityTarget* implementeert, wordt opgemerkt door de senses class. Deze class geeft vervolgens aan de AgentBase behaviour class aan over welke possibilites geïtereerd kan worden.

Figuur 1: Venn diagram dat de relatie tussen omgeving, sensoren en acties aangeeft.

1.
The list of possible actions for an agent are determined through observing the environment.

Fig 2: Diagram of possible actions for an agent



Weights

Een belangrijke eigenschap van mijn utility systeem is dat het agents een aanpasbaar ‘karakter’ geeft. Dit doe ik door utilities een ‘gewicht’ mee te geven. Een gewicht moet je in mijn framework zien als een prioriteit. De gewichten van iedere agent in mijn framework zien er als volgt uit. De volgende drie weights geven aan hoeveel waarde een agent hecht aan het **behouden en verkrijgen van:**

- Health weight
- Energy weight
- Attack power weight

Vervolgens is er een weight die bepaalt hoe aanvals gericht de agent is.

- Damage dealing weight

De utility berekening

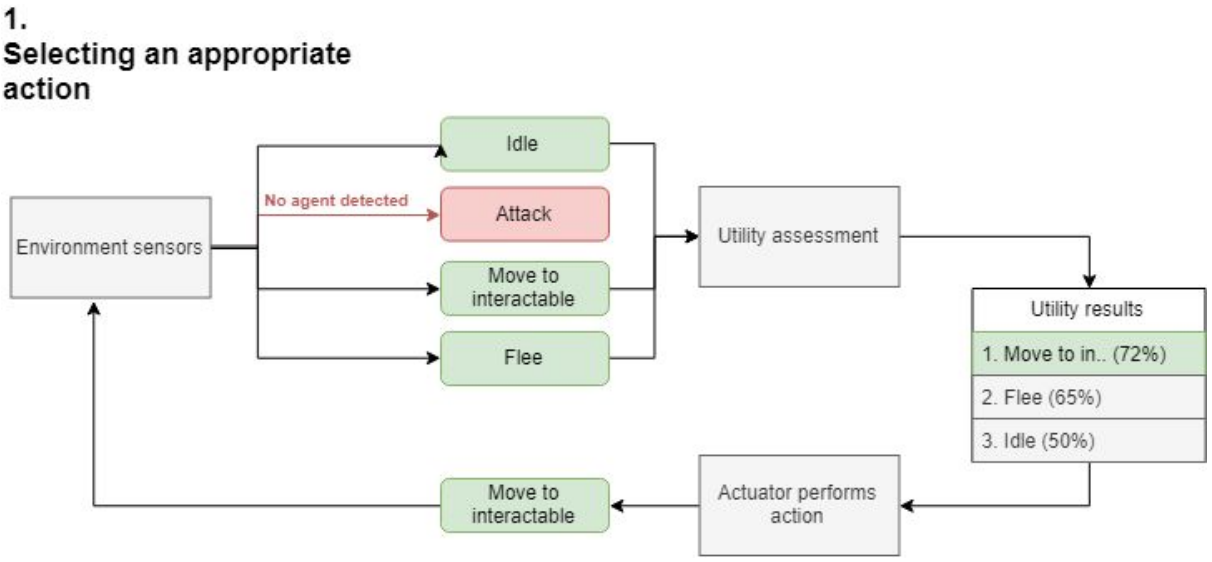
De utility berekening die op de possibilites wordt losgelaten werkt als volgt. Ten eerste wordt een possibility omgezet in een *possibility mode*l. Dit model maakt het berekenen van de utility makkelijker, omdat het alleen cruciale informatie bevat over de possibility. Een model bestaat uit de volgende properties:

- PossibilityType - het type van de possibility.
- TargetItem of TargetObject - het object waar de agent een interactie mee aan kan gaan.
- TargetPosition - de positie waar de agent zich naartoe zal moeten navigeren om deze possibility te voltooien.
- EnergyCost - de mogelijke energie kosten om deze possibility uit te voeren.
- TargetAgent - optioneel, gebruikt wanneer de possibility te maken heeft met een andere agent (i.e. aanvallen)

Vervolgens wordt met dit possibility model het utility percentage berekend. We berekenen ten eerste de volgende negen variabelen.

- **Projected health, projected energy en projected attack power**
    - Mocht de actie volgens plan verlopen, dan is dit het resultaat daarvan, uitgedrukt in health, energy en attack power.
- **Health increase, energy increase, attack increase**
    - Mocht de actie volgens plan verlopen, dan is dit het verschil tussen de projected health, energy en attack power ten opzichte van de oorspronkelijke waardes.
- **Weighted health increase, weighted energy increase, weighted attack increase**
    - De bovenstaande drie waardes multiplied met het gewicht dat aan deze increase hangt.

Figuur 1. Schematische weergave gemaakt in Draw.IO dat het itereren over mogelijkheden weergeeft.



Utility berkening

Stel, we berekenen de utility waarde van het consumeren van een item.

Agent:

Health: 80 HP
Energy: 80 EN
Attack: 5 AT
Health weight: 0.8

Agent weight set:

Health weight: 80%
Energy weight: 60%
Attack weight: 50%

Item:

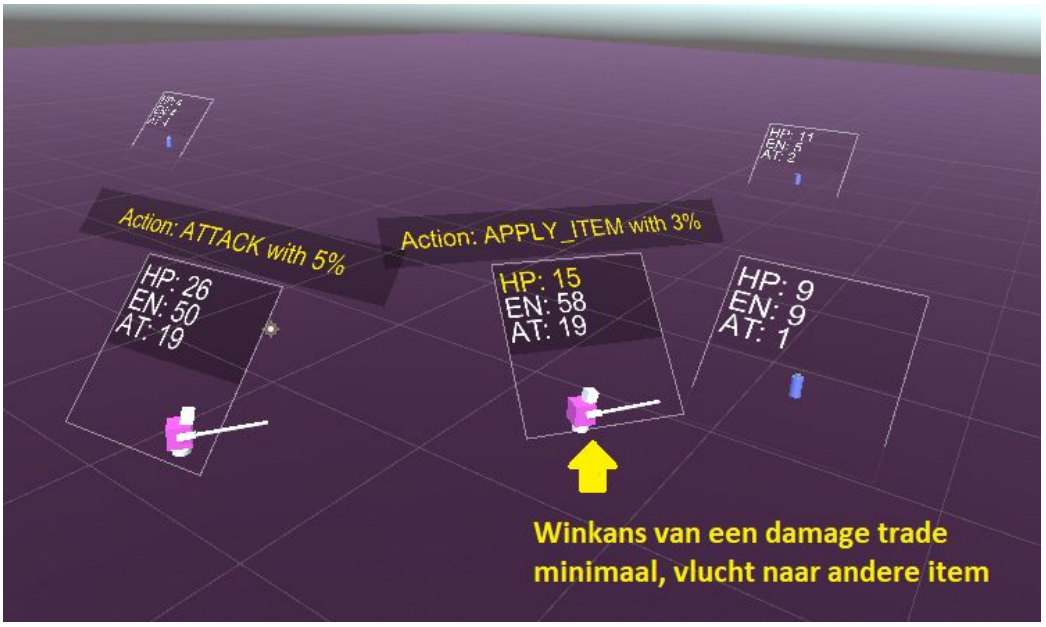
Health boost: 5 HP
Energy boost: 5 EN
Attack boost: 0 AT

Projected health: 80 + 5 = **85 HP,** increase van **5**
Projected energy: 80 + 5 = **85 EN**, increase van **5**
Projected attack: 5 AT, increase van 0

Weighted health increase: 5 * 0.8 = **4**
Weighted energy increase: 5 * 0.6 = **3**
Weighted attack increase: 0

Utility: (Weighted health incr.. + weighted energy incr.. + weighted attack incr..) / 3
Utility = 2.3%

Screenshot: geobserveerd gedrag van een defensieve agent die weet dat zijn volgende damage trade niet te winnen is (te zien aan het verschil in attack power en HP)

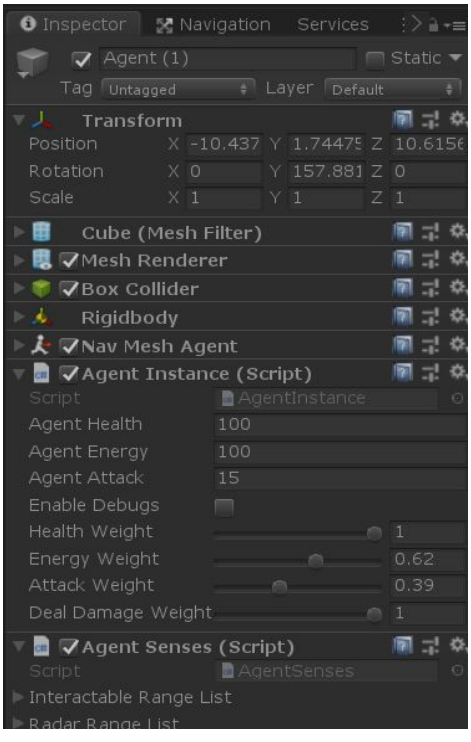


Technische opbouw

Het utility systeem is geschreven in C# binnen de Unity game engine (**2018.3.1)**. De set-up van het project is vrij simpel. Het draait om een aantal objecten die reeds in de scène geplaatst zijn.

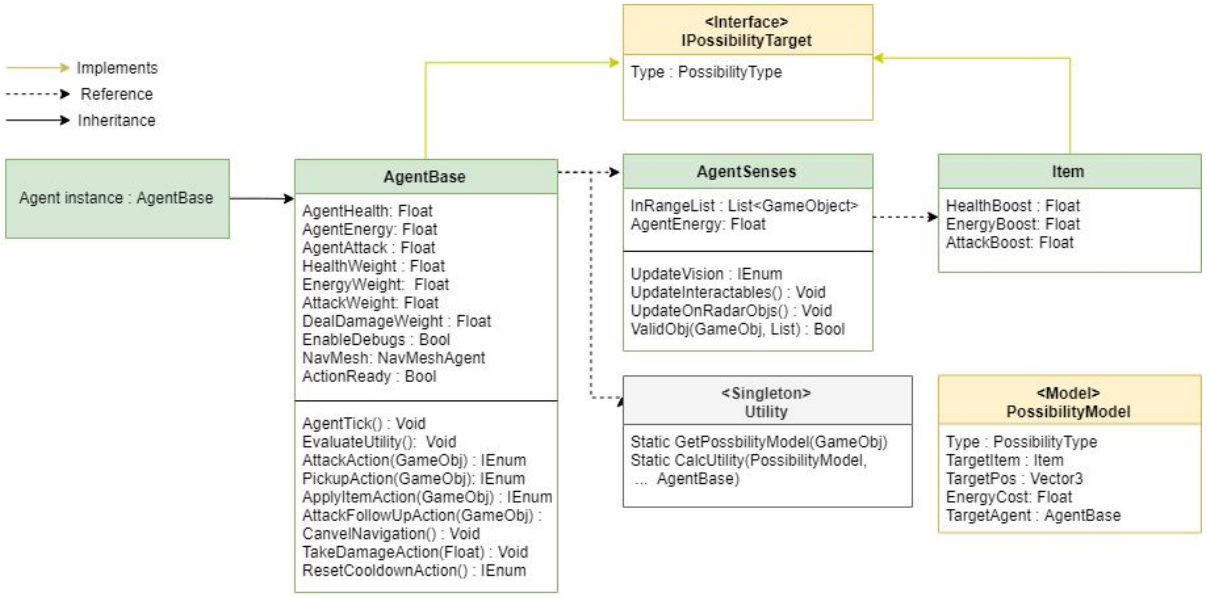
- **Agent:** De agent is een GameObject met alle benodigde classes om het utility systeem toe te passen.
    - NavMesh: verzorgt de navigatie voor de agent m.b.v. A* pathfinding.
    - AgentInstance: Instance class, inherit alles van AgentBase en kan gebruikt worden om functies uit deze base class te overriden en custom logic toe te voegen.
    - AgentBase: Bevat alle interne acties van een agent.
    - AgentSenses: Geeft aan AgentBase door welke *possibilities (*oftewel mogelijke acties) zich bevinden rondom de agent door middel van een 'radar' die met een interval een omgeving doorzoekt.
    - AgentDebug: Optioneel. Verantwoordelijk voor het *world space* canvas dat de health, energy, attack power en huidige utility actie met percentage weergeeft boven de agent.
- **ItemSpawner:** De spawner die een *Item* prefab spawnt. Items komen in de vorm van consumable acties die ieder een eigen effect hebben op de status van de agent.
    - Binnen de spawner kunnen de maximale waardes van de spawnable items beperkt worden.
- **Item:** De interactables die gebruikt kunnen worden door een agent. Iedere item kent zijn eigen waardes in:
    - Health boost
    - Energy boost
    - Attack power boost

Screenshot: Laat zien hoe de inspector view van de agent is opgebouwd. De weights kunnen in runtime aangepast worden, en iedere utility wordt boven de agent weergegeven in debug mode.



UML Opbouw van het project

Dit is de schematische weergave van alle scripts en hun onderlinge verbinding.



De codebase

De code is vrijwel allemaal gecomment, en te vinden in deze repository. Ik zet hier kort uiteene wat de belangrijkste elementen zijn en wat hun verantwoordelijkheid is.

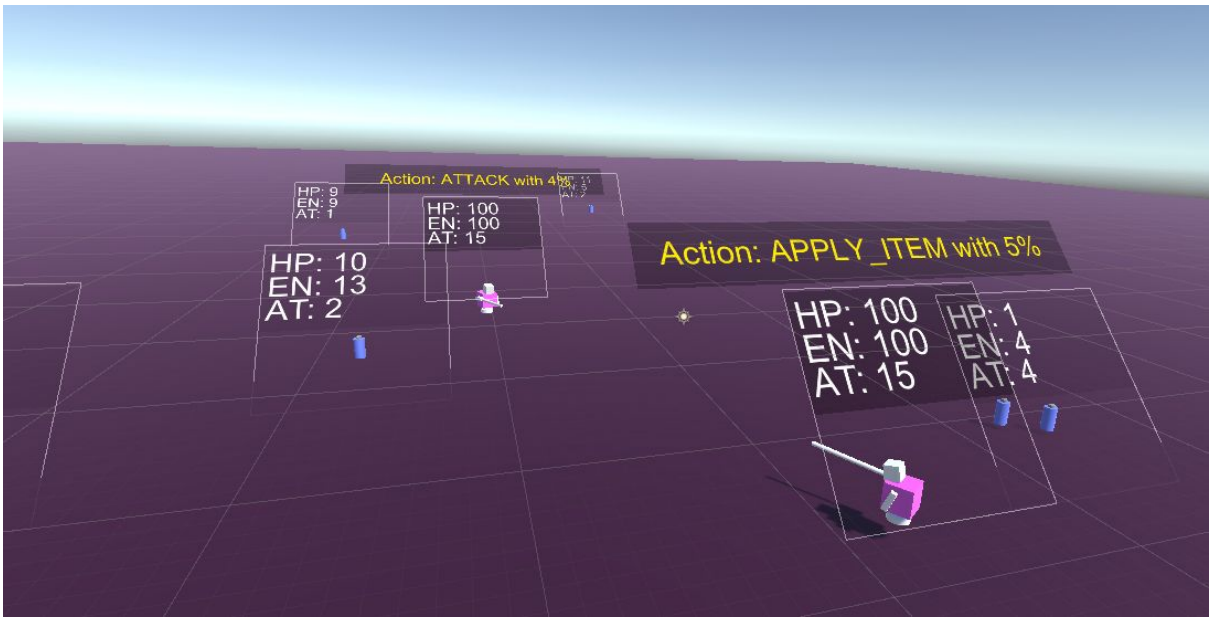
- **AgentBase**
  - De class met alle logica die benodigd is om een actie uit te voeren en alle individuele eigenschappen van een agent (beoogde gedrag).
- **AgentSenses:**
  - De class die bepaalt welke acties een agent in zijn omgeving kan uitvoeren (i.e. een item zoeken en gebruiken, een agent aanvallen).
- **Item:**
  - De class die de eigenschappen van een bruikbare item weergeeft. Denk hierbij aan mogelijke boosts voor een agent.
- **Utility:**
  - De class met twee static methods die eerst het PossibilityModel opstellen bij een object en vervolgens bepalen wat de Utility hiervan is.
- **PossibilityModel:**
  - De model class die voor iedere actie opgesteld wordt. Deze wordt gebruikt bij het berekenen van een Utility percentage.
- **IPossibilityTarget:**
  - Interface dat aangeeft dat een item een mogelijke interactie vormt voor een agent. Wordt gedetecteerd door **AgentSenses** en contracteert een type mogelijkheid.

De spelregels

Twee of meerdere agents gaan met elkaar de strijd aan in het speelveld. Ze hebben ieder een langwerpig wapen. Een agent door als laatste in leven te blijven in het speelveld.

De complexiteit zit hem in de items. Items liggen willekeurig verdeeld over het speelveld, en geven ieder een eigen hoeveelheid levens, energie en/of attack power. Iedere aanval kost energie, en de schade die de agent doet wordt bepaald door de attack power.

Iedere tick van de agent wordt per item en enemy bepaald wat de beste manier van handelen is, in percentage per *possibility.* Het zorgt voor enorm complexe en interessante handelingen onder de agents. Er ontstaat soms tijdelijk groepsgedrag.



Met vier of meer agents wordt het spel een stuk complexer. Je kan dan ook bijzonder gedrag observeren, bijvoorbeeld dat van drie agents die systematisch steeds met z'n allen de agent met de laagste health aanvallen.

Zelf aan de slag binnen het project

Het meest interessante aspect is namelijk het tweaken van de gewichten van één of meerdere agents in het speelveld. De inbegrepen build laat een statisch gevecht tussen vier agents zien, maar de gewichten en hoeveelheid agents pas je natuurlijk aan in de editor.

In het project

Je vindt twee pre-made scènes.

- **Group Battle -** Een demonstratie van vier agents die het tegen elkaar opnemen in een veld met wat willekeurige obstakels en met een random hoeveelheid bruikbare items.
- **1 Versus 1 Battle -** Een demonstratie van een 1 op 1 gevecht tussen twee agents, met willekeurige obstakels en een random hoeveelheid bruikbare items.

Agent gedrag bepalen
Is heel eenvoudig. Op iedere agent zit een **AgentInstance** class. In de inspector kan je hier met sliders het gedrag bepalen van een agent. Zo kan je bijvoorbeeld:

- Startwaardes bepalen van een agent (health, energy, attack power)
- Console debugs aan en uit zetten
- Health weight, Energy weight en Attack weights aanpassen
- Deal damage weight aanpassen

Wanneer alle weights op 1 staan, houdt deze geen rekening met voorkeuren bij het bepalen van een gepaste actie. Er wordt dan alleen gekeken naar de actie die zo veel mogelijk levens van de andere agents kost, en deze agent zoveel mogelijk oplevert.
Geef je echter een health weight een veel grotere waarde dan de andere weights, dan verkiest de agent vluchten boven aanvallen of het vinden van meer slagkracht.

Emergent behaviours:

- **Afwachtende agent:** In groepsgevechten loont het zich om een agent defensief te laten spelen. Ofwel om health het zwaarste te laten wegen. Wat er gebeurt is dat deze agent zich afzijdig houdt, en dat andere agents met elkaar in gevecht gaan. De defensieve agent maakt vervolgens de al verzwakte agents af.
- **Slagkrachtige agent:** Een agent die zich volledig focust op het verkrijgen van meer aanvalskracht wordt, mits deze niet in de eerste paar seconden verslagen wordt, vrijwel onverslaanbaar. In de late game is deze agent zo sterk dat andere agents weten dat ze een 1 op 1 gevecht niet gaan winnen. Vaak zijn er dan echter te weinig items voor deze zwakkere agents om op dat niveau te komen, waarna ze vernietigd worden door de agent.
- **Agressieve agent:** Door het DealDamage (Agressie) weight hoger te leggen dan de rest, krijg je een agent die het 1 op 1 goed doet, maar in groeps gevechten vaak overmoedig wordt en als een van de eerste sneuvelt

Dit zijn slechts enkele geobserveerde behaviours van een agent. Omdat er zoveel wisselwerking tussen weights plaatsvindt, zijn er ongetwijfeld nog veel meer interessante handelingen te observeren.

Nawoord en conclusie

Gedurende dit project werd het aanpassen van het gedrag van een agent steeds gemakkelijker. Ondanks dat de spelregels simpel zijn, kan je eindeloos variëren met het gedrag van een agent. Ik heb agents laten vechten die agressief, extreem defensief of heel hybride gedrag vertoonden.
Deze aanpasbaarheid is heel waardevol bij het balanceren van een game. Daarnaast is het toevoegen van meerdere items of aanvalstechnieken in het framework erg makkelijk, ze hoeven namelijk alleen maar het IPossibilityTarget interface te implementeren.

Machine learning toepassing
Bovendien kwam ik na dit project op het idee om de optimale agent te creëren door middel van een stukje machine learning. Dit valt buiten de scope van deze herkansing, maar lijkt me alsnog enorm interessant om te proberen. Ik ga dit als volgt aanpakken.

1) Ik maak een Agent model, waarin ik opsla:
    a) Health weight, *float*
    b) Energy weig ht*, float*
    c) Attack power weight, *float*
    d) Attack aggression, *float*
    e) Win/lose ratio a.d.h.v. een serie gevechten in een niet-willekeurige omgeving.
2) Ik laat vervolgens agents met willekeurige gewichten duizenden rondes spelen, en sla alle models inclusief winratio op.
3) Ik kies steeds de best presterende agent (o.b.v. win/lose ratio), en laat deze tegen variaties van zichzelf spelen totdat het win/lose ratio niet of nauwelijks meer stijgt.

Screenshot in IDE: Laat de code-behind zien van de utility berekening van een possibilty.

```
// Calculate the percentage of
// energy gained from this item
projectedEnergy = agent.AgentEnergy + model.TargetItem.EnergyBoost;
if(projectedEnergy > 100)
    projectedHealth = 100;

energyIncrease = projectedEnergy - agent.AgentEnergy;

// Calculate the percentage of
// attack power gained from item
projectedAttack = agent.AgentAttack + model.TargetItem.AttackBoost;
if(projectedAttack > 100)
    projectedAttack = 100;

attackIncrease = projectedAttack - agent.AgentAttack;

// Correct our gain with weights
// that determine the agent's priorities
weightedHealth = healthIncrease * agent.HealthWeight;
weightedEnergy = energyIncrease * agent.EnergyWeight;
weightedAttack = attackIncrease * agent.AttackWeight;

// Calculate the utility
float utility = (weightedHealth + weightedEnergy + weightedAttack) / 3;

// Debug the utility
if(agent.EnableDebugs)
    Debug.Log("Agent: " + agent.name + " - Calculated utility of: " + utility + " % "
    + "for an item");

return utility;
```