Complex System Project Brief

Unity Modular Utility AI

* the purpose of the system,
  + To provide an easy-to-use Unity package that can be imported into any Unity project and used without the developer needing much knowledge of the AI systems behind it.
  + The system will run almost completed on delegates and unity events, which allows developers to add any call back, evaluation, or condition methods through the inspector or code.
  + The goal is to give the developer control over how behaviours are evaluated by allowing them to provide methods that calculate values for their game situations; the conditions that interrupt behaviours; or what happens when a behaviour starts, is active, or has ended. They will not need to know how behaviours are stored and checked or how the subsequent methods are called.
* any libraries it relies on,
  + The system only relies on the in-built Unity systems.
* the mathematical operations to be used,
* the advanced algorithms and systens to be implemented,
  + The Unity editor system does have an inbuilt system for displaying delegates in the inspector. There is support for Unity events, however these do not allow for return types will are necessary for the evaluators, conditions, and behaviour selectors. To make it possible for the user to edit these delegates in the inspector, a custom Unity editor drawer system will have to be made. Through the utilisation of a container class and reflection, the editor will be able to take an object, display all of its methods in a dropdown, and allow for the creation of a delegate on the backend using a chosen method name.
* how it will be made modular, and
  + The system will be made modular using the Unity inspector to dynamically generate Utility AI classes.
  + The developer will be provided with the tools to generate as many behaviours as they’d like all from the inspector.
  + The generation systems behind the scenes will generate a brand-new C# script with the Utility AI systems, as well as new serialised variables for each custom behaviour chosen by the developer. The developer can then use a script that inherits from this newly generated script to access the behaviours and alter their functionality.
* how to integrate your system with a new or existing application.
  + Import the Modular Utility AI package into your Unity project.
  + Create a new script which inherits from the *UtilityAI* class
  + Return to the Unity editor and attach your newly created script to an object such as an enemy.
  + In the inspector, your script will have a behaviour list and a generate button. Fill the list with any number of behaviours you’d like. (Behaviours can be added, updated, or removed at any point in the future)
  + Once behaviours are chosen, press the Generate AI Instance button. This will create a new Utility AI behind the scenes. Your script should now inherit from this newly generated script, and include an interface which provides AI Awake, Start, and Update methods.
  + You will now see each of your behaviours as categories in the script inspector,

# Images example

User makes a new script called “AITest” and inherits from the “UtilityAI” class.

A picture containing text, screenshot, font, software

Description automatically generated

In the inspector they are presented with the instance generator. They can add as many behaviours they’d like.

A screenshot of a computer

Description automatically generated with medium confidence

After adding the behaviours they’d like to generate, they click the Generate AI Instance button.

A screenshot of a computer

Description automatically generated with medium confidence

This will automatically create a new script with the name of their script (“AITest” in this case) prefixed with “UtilityAI\_”. Their script will now inherit from this new script and a “IUtilityAIMethods” interface which will prompt them to add the AIAwake, AIStart, and AIUpdate functions.

A screen shot of a computer program

Description automatically generated with medium confidence

Now if they look at the inspector again, they will be presented with all their chosen behaviours as serialised fields.

A screenshot of a computer

Description automatically generated with medium confidence

Each behaviour has many fields which can be edited. These include:

* The evaluation cooldown: The time between each re-evaluation of the behaviour score
* Value range min and max: The range of values which are expected to be returned by the evaluator (E.g. the user expects their run evaluation to return a value from 25-80. This will be remapped behind the scenes to a value between 0-1.)
* The evaluator delegate method: A method which returns a float value of a behaviour’s score (E.g. a value from 25-80 on how much an enemy wants to run away from the player)
* A timer duration: A float which is used by a timer condition to deactivate a behaviour after a certain time (This condition will be an inbuilt method of every UtilityAI which can be added in the Interrupt Conditions – behaviours do not require a timer condition)
* Interrupt Conditions: A list of methods which return true or false to whether a certain condition should end. If any condition becomes true, the behaviour can end (e.g. an enemy has a run behaviour and a tiredness condition, and if the enemy becomes too tired it will look to do a new behaviour)
* When Active: A list of methods to be called every frame while a the behaviour is active
* On Start: A list of methods to be called when a behaviour first begins
* On End: A list of methods to be called when a behaviour ends

A screenshot of a computer

Description automatically generated with medium confidence

The AI system also has a behaviour selector which is a method that takes in an array of floats representing behaviour scores, and returns an index of any of those floats to be the new chosen behaviour. The UtilityAI class will have a few inbuilt methods or the user can make their own.

