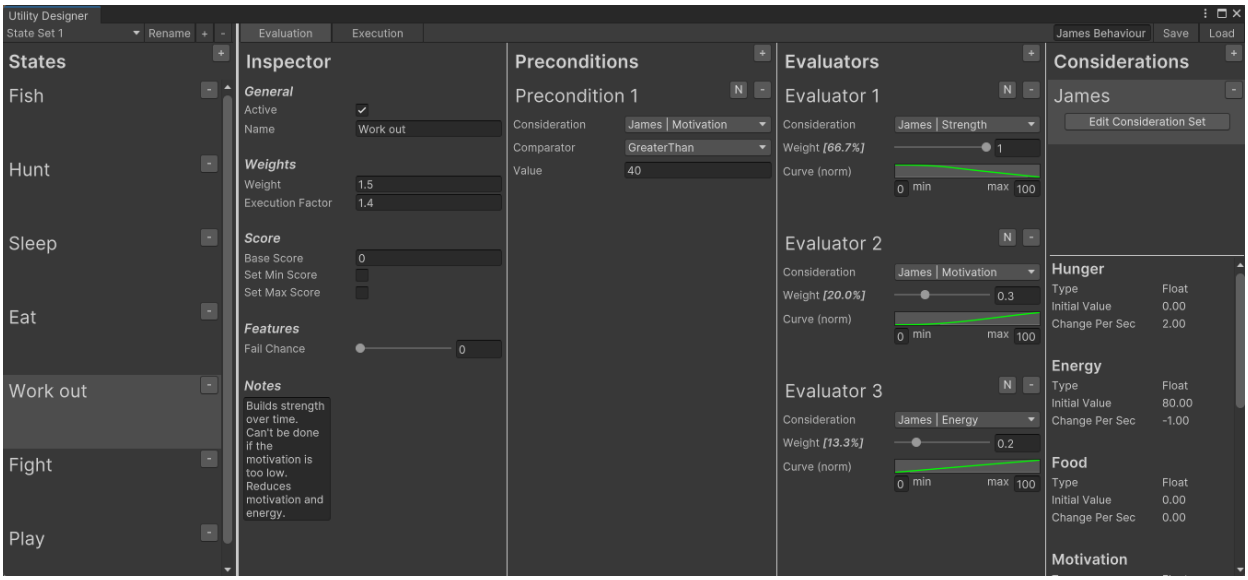
Utility Designer

Manual



Version: 1.0.0

Contents

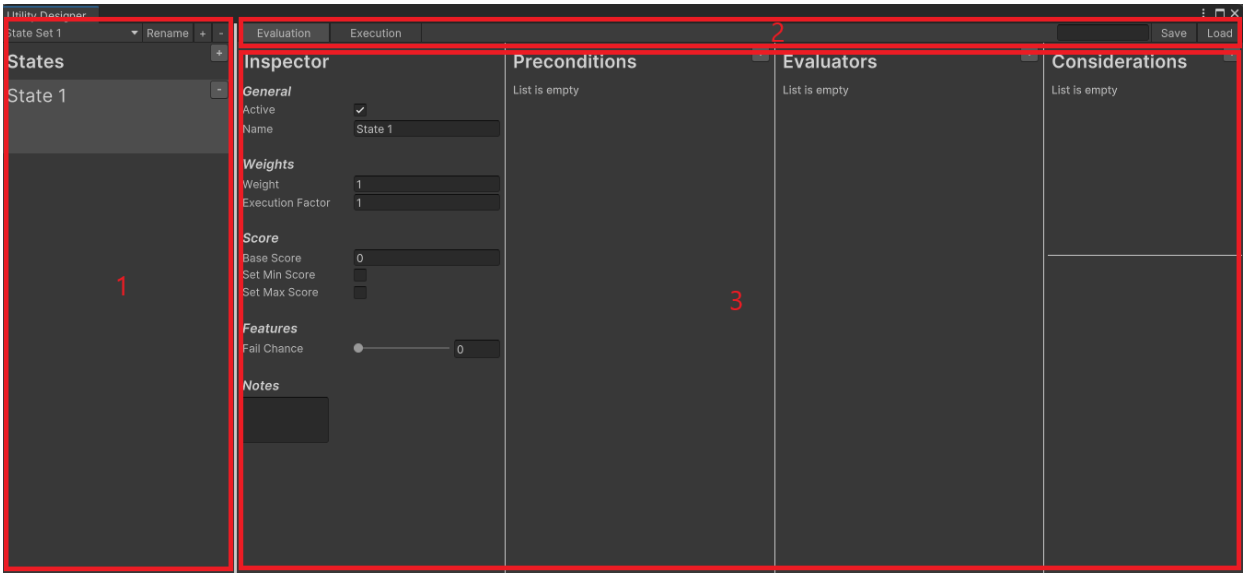
Utility Designer

Overview

Utility Designer combines utility AI and behaviour trees, making it easy to create intelligent and dynamic Non-Player Characters (NPCs) or other types of behaviour. Utility AI is used to decide which high-level state to execute, with each state having an associated behaviour tree that defines what exactly that state does. As Utility Designer provides an implementation of a behaviour tree, it could alternatively be used as a normal behaviour tree, using only one state and thus ignoring the utility AI part.

Getting started

The first step is to ensure that the asset is correctly imported into the project. To create a new behaviour, the Utility Designer script must be added to the required object. The editor window is split into different sections and when you first open the window and add a state, it will look like this:



Section 1: This lists all the states that the NPC can choose from. New states and state sets can be added and removed at will. Each state has its own inspector, preconditions, evaluators and a behaviour tree.

Section 2: **Utility Designer consists of two tabs: one for evaluation (utility AI) and one for execution (behaviour tree).** The tab can be selected on the left side. On the right, changes can be linked to a scriptable object, which will automatically save any further changes.

Section 3: This section displays the currently selected tab. In this example, the Evaluation tab is displayed for editing.

Utility Designer Script

The Utility Designer script contains a Utility Behaviour scriptable object which defines the behaviour of the NPC and can be edited using the 'Open editor' button. The script also provides an interface to reference the *SceneReferences* and change the frequency of which the Evaluation and Execution tabs are updated. The lower the tick rate, the faster the NPC can react to changes, but at the cost of performance. Both tabs are independent and can have different tick rates. Alternatively, Unity's Update method can be used as the tick rate, which corresponds to the end user's frame rate.

In the 'Evaluation' section there is another toggle called 'Log To File', which logs all run-time evaluation to a .txt file in the Assets folder. This file will be overwritten every time the scene is run, but it will only log while the scene is running in the Unity editor.

Note that a GameObject can only have a maximum of one Utility Designer script attached to it.

State

In the context of Utility Designer, the NPC is always in exactly one state. Each state represents a specific high-level goal that the NPC can choose from. The Evaluation tab allows you to configure how the NPC decides which state to choose, while the Execution tab specifies what the state does when executed.

For instance, in a game, an NPC's states might include 'drinking', 'eating', 'sleeping', 'exercising' or 'playing'. Each of these states would have associated behaviours and reactions that define what the NPC does when it's in that state.

State Set

A state set in Utility Designer is a collection of states from which an NPC can choose. The concept of a state set allows for greater organisation and flexibility in designing NPC behaviour.

For example, one can create separate state sets for different situations or contexts in the game. An NPC might have a 'Patrol' state set for when it's on regular patrol, a 'Fight' state set for when it encounters an enemy, and a 'Rest' state set for when it's idle or recovering.

As the NPC's context changes, it's possible to switch between these states with the action node 'Change State Set'. This allows for a wide variety of potential behaviours for the NPC, depending on their current situation, and to react dynamically to changes in its environment, allowing them to behave in a more complex and realistic manner.

Evaluation tab

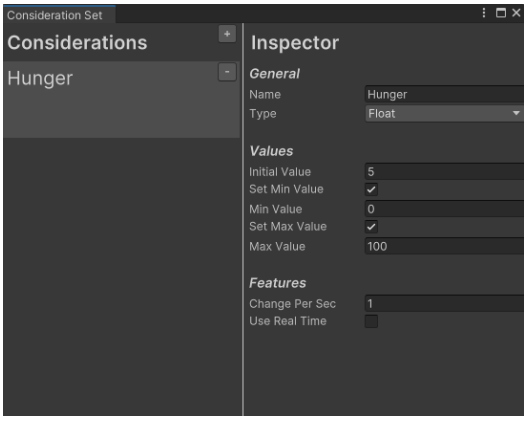
What is Utility AI

Utility AI can be thought of as a pool of states, with the best state being decided and executed at all times. In a utility AI system, each potential state is given a utility score based on various factors, such as the current context, goals, or needs of the character. These scores represent the desirability or usefulness of each state. At each evaluation tick, the state with the highest score is selected and its behaviour tree is executed.

For the NPC to know about their environment, considerations are used, which serve as the NPC's perception. These considerations can change over time, allowing the NPC to respond effectively to changes in its environment. Each state can then choose which considerations it wants to take into account with the help of evaluators. For example, the state 'Eat' could be interested in the considerations 'Hunger' and 'Money', allowing this state to adjust its score based on these two factors.

Consideration

Consideration is a utility AI specific term, where a consideration represents a specific piece of information that the NPC can use to decide which state to choose. They are like variables that can be changed from inside or outside the utility AI system.



This shows a consideration representing an NPC's hunger. By default, a consideration can take any value that its type can take, but the optional 'Set Min Value' and 'Set Max Value' toggles allow it to be constrained. This is useful if you want the consideration to represent a percentage, such as hunger from 0% to 100%.

When the scene is loaded, the 'Initial Value' is assigned to the consideration, and while the scene is running, the value automatically changes according to the 'Change Per Sec'.

Consideration Set

As the name suggests, a consideration set is a set of considerations grouped together so that they can be easily added to the Utility Designer editor. Consideration sets are scriptable objects, and a new set can be created by right clicking in the Projects folder, then 'Create → Utility Designer → New Consideration Set'. The editor can be opened by double clicking on the scriptable object. New considerations are added using the plus button at the top and there is no limit to the number of considerations a set can hold.

To use consideration sets in the Utility Designer editor, they must be added from the Evaluation tab on the right. Once added, they can be opened and edited directly from the Utility Designer editor.

Local Consideration Set

By default, a consideration set is global, which means that changing the value of a consideration will change it for every *UtilityBehaviour* using that consideration set. This is useful for global events that are the same for every NPC, such as weather, time, environment, etc.

However, there are some considerations that every NPC of the same type has, but their current values are specific to each one. These include hunger, energy, thirst, mood, etc. When a consideration set is set to 'Local', it is unlinked from the *ScriptableObject* in play, so any changes made to the values of its considerations will stay local and not be passed on to other NPCs using that consideration set.

This allows NPCs with a *UtilityDesigner* component to still have independent consideration values, even after they have been copied and pasted or instantiated from the same prefab.

Accessing Considerations

Because considerations represent the knowledge of the NPC of their surroundings, it's important to keep them updated at any time. This can be achieved either through an action node in the behaviour tree or directly by code using the API of the consideration set.

A *ConsiderationSet* has three public methods, and adding a reference to a specific consideration set in any script allows accessing them:

- *bool* ***SetConsideration****(string considerationName, float newValue)*
- *bool* ***ChangeConsideration****(string considerationName, float amount)*
- *float* ***GetConsideration****(string considerationName)*

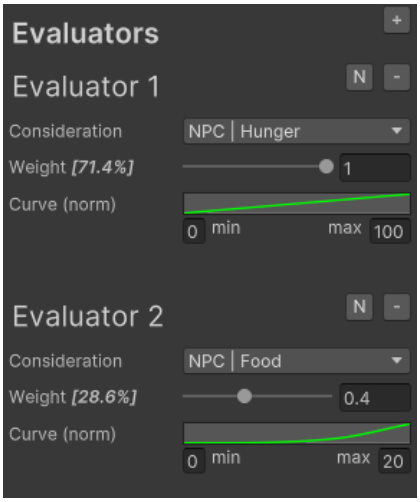
With these three methods, considerations can either be set to a certain value, changed by a specific amount, or its value can be retrieved. Considerations are always accessed with their name as a string. Be aware of that when renaming considerations.

Local considerations need to be accessed with an additional parameter to get a reference to its *UtilityDesigner* script. This parameter can either be of type *UtilityDesigner*, or GameObject. Keep in mind that the GameObject needs to have a *UtilityDesigner* script attached, and it's less performant, as it uses *GetComponent<>* to reference the *UtilityDesigner* script.

Evaluator

Evaluators are used to determine the utility score for each state. Each evaluator is tied to a consideration and uses it to score each state based on the current context of the NPC.

Each evaluator uses a curve to translate the value of a consideration into a utility score. The curve can be adjusted to capture the NPC's preferences, or the desirability of different states given the value of the consideration. Also, each evaluator has a weight that determines how much the score from the curve will contribute to the total state score.



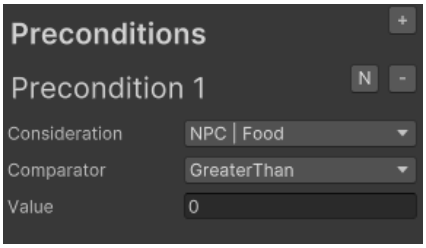
Here, two evaluators are used to score the state 'Eat'. The hungrier the NPC is and the more food they have in their inventory, the higher the score for this state and the more likely they are to eat.

For example, the consideration 'Hunger' currently has a value of 80. This translates into a score of 0.8 based on the linear curve used by the first evaluator. This score is now multiplied by the weight, in this case times 0.714 (71.4%), which means that this evaluator currently contributes a total of 0.8 * 0.714 = 0.5712 points to its state.

If 'Hunger' changes, the score will change accordingly. The same calculation is done each evaluation tick for every evaluator.

Precondition

A precondition is a requirement for a consideration that must be met before a particular state can be considered for evaluation. Under no circumstances will a state be picked unless all of its preconditions are met. Preconditions are checked in each evaluation cycle. If a state's preconditions aren't all met, that state is excluded from the current decision process and its utility score is not calculated.



This example shows a precondition that checks whether the NPC has food in their inventory or not. If they don't have any food, then there's no point in evaluating this state, and the precondition is considered not satisfied.

Preconditions can be used to ensure that the NPC doesn't consider states that are currently impossible.

Inspector

Each state has an associated inspector that allows further customisation of the state.

An example state called 'Eat'.

The 'Active' toggle can be used to disable the state when it's not needed, such as during test scenarios.

'Weights' is a factor applied to the final score to express the importance of a state. The "Execution Factor" is also applied to the final score, but only if the state is currently being executed. This helps the NPC stay in a state for a while, rather than constantly switching between states.

The 'Base Score' is added to the total score before the weights are applied and represents a certain base desirability for the NPC for that action. 'Set Min Score' and 'Set Max Score' enable the option to clamp the final score after multiplying by the weights, preventing actions from not scoring too high, for instance.

Whenever a new state is chosen, there's a 'Fail Chance' that the next best state will be chosen instead, which works recursively. This can help to mirror the mistakes that humans make.

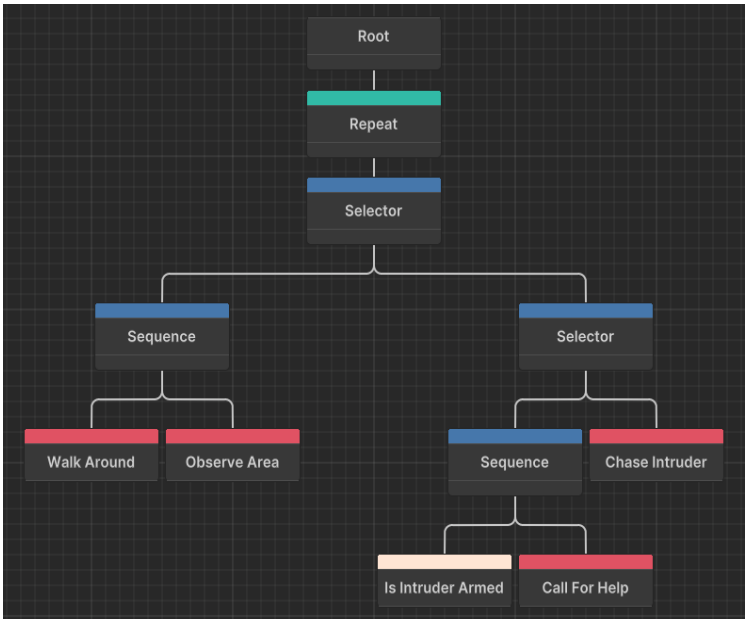
Execution tab

What is a Behaviour Tree

At its most basic level, a behaviour tree breaks down complex actions into smaller, simpler tasks. Each task is represented by a node in the tree. It starts at the root node and moves down the tree, making decisions based on the status of each node: *Success*, *Failure* or *Running*. The tree executes its nodes from top to bottom and left to right, following a set of rules to decide which node to execute next.

There are four types of nodes in the Utility Designer behaviour tree:

- **Composite:** Defines the base rule for how the branch is executed. (x children)
- **Decorator:** Able to modify the return value of a node. (1 child)
- **Action:** Contains the logic to be executed when it's invoked. (no children)
- **Conditional:** Makes Boolean decisions based on the defined condition. (no children)



This is an example of an agent patrolling an area. The first node after the root node is the “Repeat” node, a decorator. This node always returns *Running*, so it keeps on executing the child node.

The next node is the 'Selector' node, a composite. It starts by executing the leftmost child node and waits for it to finish, in other words, it waits for the 'Sequence' node to return either *Success* or *Failure*. A 'Selector' works like an OR operation. This means that once the first child returns *Success*, it will also return *Success*. If the child returns *Failure*, it will execute the next child node. If all child nodes fail, it will also return *Failure*.

The 'Sequence' composite is the opposite of the 'Selector' node and works like an AND operation. It will continue to execute all child nodes, until one returns *Failure*, at which point it will abort and return *Failure*. If all child nodes succeed, it returns *Success*.

The 'Sequence' on the left will now make the agent first execute 'Walk Around' and then 'Observe Area' once he has walked around for a while. This process will repeat until one of these two action nodes fails, at which point the first 'Selector' will execute the right sub-tree, meaning that an intruder has been detected.

In there, the agent first checks whether the intruder is armed with the conditional node, and if so, he will call for help. If not, 'Is Intruder Armed' returns *Failure*, which causes the 'Sequence' to also return *Failure*, and the 'Chase Intruder' action is executed instead.

Scene References

When performing various actions, nodes often need to interact with or affect objects in the scene. However, action nodes in the Utility Designer tool do not inherit from *MonoBehaviour*, which means that they lack the built-in field that normally allows direct association with a scene object. To work around this, the Utility Designer uses a mechanism called *SceneReferences*. This allows you to access objects in the scene from any part of the behaviour tree.

To create a new *SceneReferences* script, it must inherit from the abstract *SceneReferences* class. In the scene, the necessary object types to reference are registered by lists of those component types. This is done in the overridden *RegisterCustomLists* method.

Here's an example:

```
public class ExampleReferences : SceneReferences
{
    [SerializeField] private List<GameObject> gameObjects;
    [SerializeField] private List<Transform> transforms;

    protected override void RegisterCustomLists()
    {
        AddList(gameObjects);
        AddList(transforms);
    }
}
```

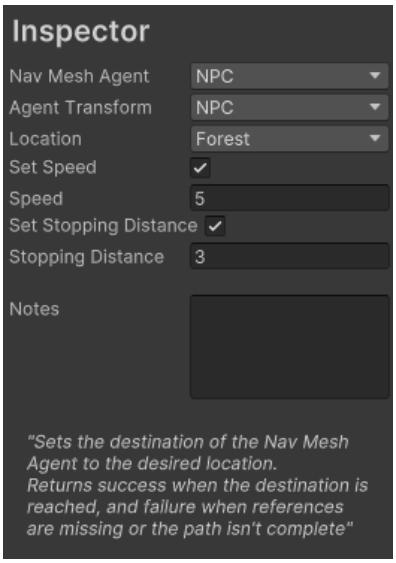
The *ExampleReferences* script needs to be on any GameObject in the scene with a unique name. In the Utility Designer script, the name of the GameObject is to be put in the 'Scene Refs Obj Name' field, so it can be found by the script. *SceneReferences* can't directly be linked to the Utility Designer script because any reference to the scene will be lost when making a prefab out of it.

The way *SceneReferences* are accessed through code in custom made action nodes is described in the 'Creating custom nodes' chapter.

The generic method *GetRef<>* allows to get any reference which has been assigned to *SceneReferences*. This can be done either by the index of the list, by the name of the GameObject or with a custom function to query for the desired reference. The whole list can also be retrieved using the generic *GetListOfType<>* method.

Node inspector

When clicking on a node, the node inspector on the right side of the window shows different elements of the node. Its main use is to learn more about the node and to set the value of serialised variables.



In this example of the Move To Transform node, there are three dropdowns linked to the *SceneReferences*. If the *SceneReferences* are missing, a warning is displayed. This node also has four serialised fields, which allow the node to be further configured..

After all the serialised fields, each node has a 'Notes' field, which allows you to make custom, persistent notes about the node's usage.

A short description of the node is given below. It is set in the code and helps to understand the behaviour better.

Creating custom nodes

Creating nodes and their call-backs

Utility Designer comes with several action nodes already implemented, which are useful for many use cases. However, as some actions require very specific behaviour, the creation of custom nodes is supported by a powerful API.

To create a custom action node, start by creating a new class that inherits from the *ActionNode* class.

The only method that needs to be implemented when inheriting from *ActionNode* is the *OnUpdate* method, as it's needed to let the behaviour tree know the state of the node.

Nodes offers four call-backs:

void **OnAwake()**	This method is called only once, when the node is first started, making it an ideal place for initialisation logic.
void **OnEnable()**	Called each time the node is enabled, which makes it great for enabling logic or resetting certain parameters.
NodeState **OnUpdate()**	It's called on every execution tick, as defined by the Utility Designer script. This is a place intended to put logic that needs to run continuously or to check conditions on every tick. It also needs to return the current state of the node.
void **OnDisable()**	Called each time the node is disabled, making it a useful for any cleanup logic. It's also called when the node is aborded while running, such as when there's a state change.

Displaying variables in the node inspector

All public variables are visible in the inspector and can be changed from there. But not every type is supported, such as GameObject and Transform, because references from the scene can't be assigned directly to the node inspector, but work through *SceneReferences*.

Here's a list of all the supported types:

int, float, double, string, bool, Enum, Vector2, Vector3, Vector4, Vector2Int, Vector3Int, Color, Rect, Bounds, Quaternion

In addition, it also supports complex objects that are neither primitive types nor enums and are not derived from UnityEngine.Object. For these complex objects, it displays a foldout in the inspector that can be expanded to show their fields.

To display values from the *SceneReferences* as dropdowns in the inspector, or to display other custom dropdowns, the *RegisterDropdown* method can be overridden. In it, a new dropdown can be registered using the

void ***AddDropdown****<T>(string nameInInspector, List<T> listType, int selectedIndex, Action<int> onSelectedIndexChanged)*

method.

Displaying variables in the node itself

To make it easier to keep track of a large behaviour tree, each node can have variables displayed directly in the node itself, without having to click on it. To achieve this in a custom-made node, the *RegisterSerlializedVariables* method can be overridden, where new variables can be added using the

void ***AddVariable****<T>(string name, T variable)*

method. Values from *SceneReferences* can also be displayed by simply retrieving the value from *SceneReferences* and using it as the second parameter.

Accessing helper methods and properties

Scene References

SceneReferences can be easily accessed from within the custom node script by simply using the protected property *SceneRefs*. This gives access to the *SceneReference* script that is on the same GameObject as the *UtilityDesigner* script.

Consideration Sets

With the method

ConsiderationSet ***GetConsiderationSet****(string considerationSetName)*

all consideration sets can be accessed by their names, which have been added to the Utility Designer.

Description

Assigning a string to the protected property *Description* causes the node to display the assigned string in the node inspector, which can be useful for understanding the functionality of the node.

Utility Designer

The protected property *UtilityDesigner* can be used to get access to its *UtilityDesigner* script. Its main usage is to access a local *ConsiderationSet* with *GetConsiderationSet*.

This Game Object

To access the GameObject the node is attached to, the protected property *ThisGameObject* can be used. This way, it doesn't need to be assigned to *SceneReferences*.

Examples

Here's an example of a simple custom action node that just waits for a fixed duration:

```
public class WaitFixed : ActionNode
{
    public override string Description => "Waits for a fixed duration and then returns true.";

    public float duration = 1f;

    private float _startTime;

    protected override void RegisterSerializedVariables()
    {
        AddVariable(nameof(duration), duration);
    }

    protected override void OnEnable()
    {
        _startTime = Time.time;
    }

    protected override NodeState OnUpdate()
    {
        return Time.time - _startTime < duration ? NodeState.Running : NodeState.Success;
    }
}
```

The *duration* variable is public and will therefore be configurable in the node inspector. The *OnEnable* method is called each time the node is executed, so this is the right place to set the start time. In *OnUpdate*, the *Running* node state is returned as long as the second is not yet over, otherwise *Success*.

Here, the node also has the description set, to explain to the user what it does. Also, the *duration* variable is being serialised with the *RegisterSerializedVariables*, which will make it show up in the node.

Another example is an action node that causes the Nav Mesh Agent to move to a particular location. In there, a dropdown with access to the Transforms of the *SceneReferences* script might be useful. Implementing this could look as follows:

```
[SerializeField] private int _locationIndex;
[SerializeField] private int _navMeshAgentIndex;

private Transform _location;
private NavMeshAgent _navMeshAgent;

protected override void RegisterDropdowns()
{
    AddDropdown("Location", SceneRefs.GetListOfType<Transform>(), _locationIndex,
newIndex => { _locationIndex = newIndex; });

    AddDropdown("Nav Mesh Agent", SceneRefs.GetListOfType<NavMeshAgent>(),
_navMeshAgentIndex, newIndex => { _navMeshAgentIndex = newIndex; });
}

protected override void OnAwake()
{
    _location = SceneRefs.GetRef<Transform>(_locationIndex);
    _navMeshAgent = SceneRefs.GetRef<NavMeshAgent>(_navMeshAgentIndex);
}
```

The two new dropdowns need to be initialised in the *RegisterDropdown* method. Looking at the first dropdown, its displayed name is 'Location', and to serialise a list from *SceneReferences*, the list can simply be retrieved with *SceneRefs.GetListOfType<>*.

It's also important to store the index of the item selected in the node inspector in the script. To make this persistent, the *[SerializeField]* attribute must be added, and it must also be passed to the *AddDropdown* method so that it knows which item is selected by index. The fourth parameter is a simple function that allows the newly selected index to be retrieved when the user changes it in the node inspector.

The actual location can then for example be retrieved in the *OnAwake* method and stored in a variable, using the *SceneRefs.GetRef<>* method.

Guidelines and Best Practices

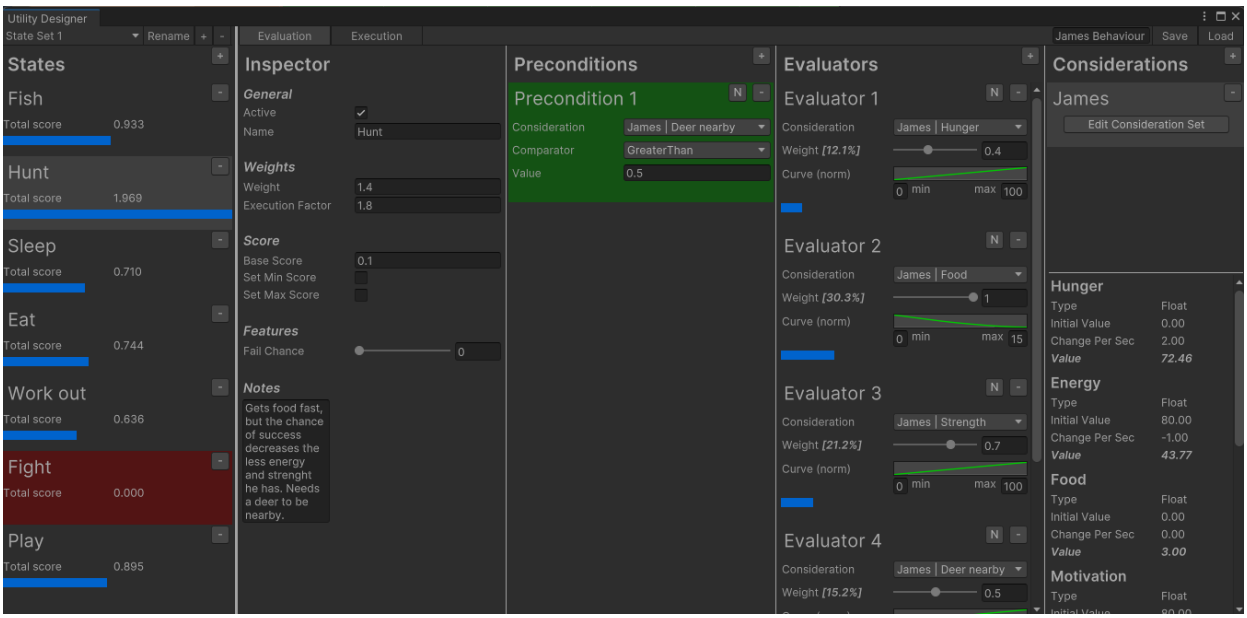
There are a few things to look out for when using the behaviour tree:

- Remember that the tree is only updated every update tick, as adjusted in the *UtilityDesigner* script. Thus, if the execution tick rate is set to 1s, and there's a node that waits for 0.1s, it will still cause the node to wait at least 1s. If timing is important in the behaviour tree, using the Update method as tick rate is recommended.$
- Single nodes can be copy pasted with CTRL + C and CTRL + V. Whole subtrees can be copied by simply selecting the parent node, not by multi-selecting all nodes.
- It's best to delete nodes from the behaviour tree before renaming them or changing their namespace, because Utility Designer will not be able to locate them afterwards.

Runtime debugging

Evaluation

In order to understand certain decisions better made by the NPC, Utility Designer displays all the scores from the evaluators and states, as well as whether the preconditions have been met, and also the current values of all the considerations.



The state scores are visualised by a blue bar. The state with the highest score is shown as 100%, and the other states are shown in proportion to this highest score. If at least one precondition isn't met, the state is coloured red.

If a precondition is green, it means its condition has been met, otherwise it hasn't.

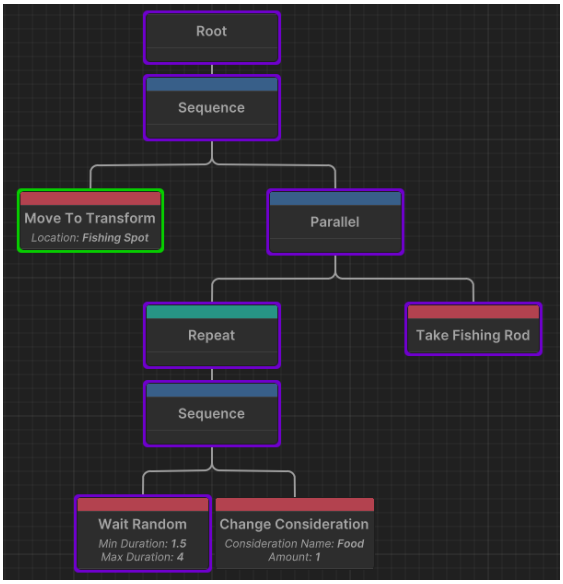
For the evaluators, they all show the score they contribute to their state with a blue bar that goes from 0 to 1.

In addition, the current value of each consideration is displayed next to the 'Value' label.

It's not recommended to change values at runtime, as some calculations are only done once at the beginning for performance reasons and changing them at runtime could lead to incorrect results. Also, any changes made at runtime will be saved directly to the Utility Behaviour scriptable object and will not be undone when you exit play mode.

Execution

The behaviour tree also provides runtime debugging, where running nodes are purple, successful nodes are green and failed nodes are red.



Resources and Support

There are three Unity packages in the 'UtilityDesigner' folder containing two sample scenes. One for the built-in render pipeline, one for URP and one for HDRP. Import the one you need and you'll find the example scenes in the 'Demos' folder. Make sure to import Unity's AI Navigation package before running the sample scenes.

Here is a video link that briefly explains the basic concept of Utility Designer:

https://youtu.be/oxTH4gS89MU

If there is a bug in Utility Designer, or if you have any questions or feature requests regarding this asset, please feel free to contact me via email:

KadaXuanwu@gmail.com

Consider taking a moment to explore the games I've made:

https://discord.gg/ZCG2ne9zVV