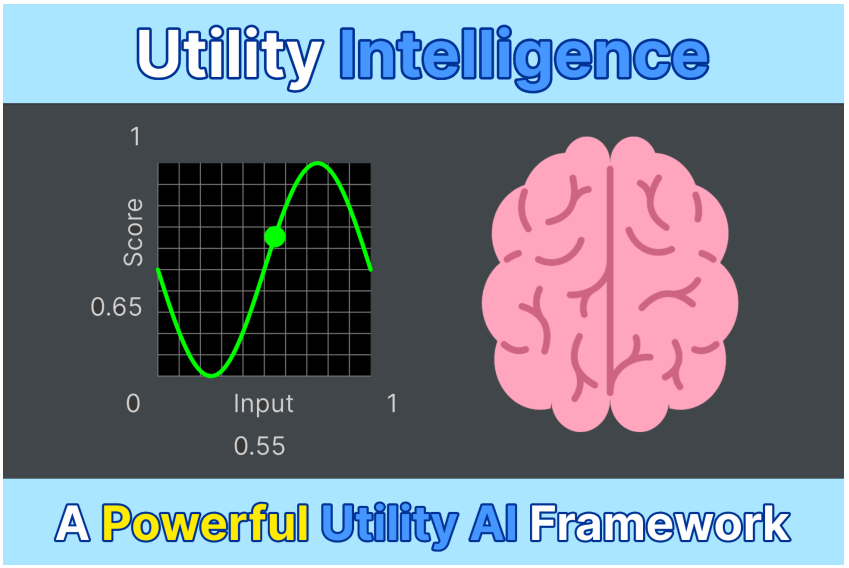


Online Documentation

Note: We don't update the offline version frequently because it's time-consuming. Therefore, if possible, refer to the online version for the best experience. It's always up to date.

Welcome to the Documentation Hub

Hope you have a great experience with Utility Intelligence!😘

Table of Contents

Tips & Tricks

Documentation

Overview

What is Utility Intelligence?

Utility Intelligence is a robust and powerful **Utility-Based AI** Framework. It allows **agents** to **make decisions** based on **scores**. Therefore, **designers** can **adjust** the **decision-making** process by **tweaking** the decision **scores**, **without** needing **support** from **developers** to **change** the **behavioral structure**, as required in **Behavior Trees** and **Finite State Machines**.
-> **Designers** and **developers** can **work independently** without affecting each other.

- **Designers**: Focus on adjusting the decision scores to ensure the best decision is chosen in any situation.
- **Developers**: Focus on creating and executing new decisions based on the game design document.

How Utility Intelligence works

Here's how **Utility Intelligence** works step by step:

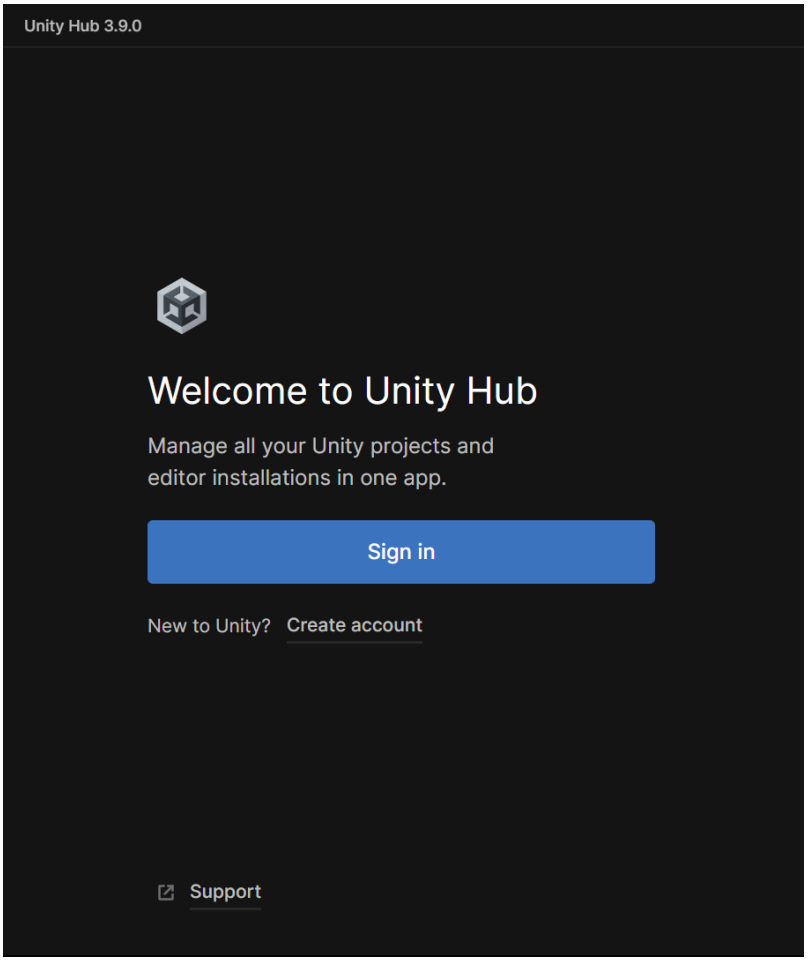
1. **Add** decisions to the agent.
2. **Score** every decision based on the current situation.
3. **Select** the decision with the highest score.
4. **Transition** from the current decision to the selected decision.
5. **Execute** the action tasks of the selected decision sequentially or simultaneously.


The best decision for this situation
Continue developing and maintaining the package at a slower pace.
0.75
Continue developing and maintaining the package at a faster pace.
1.00
Abandon the Behavior package, but keep the team because they are awesome.
0.50
Lay off the Behavior team and abandon the package.
0

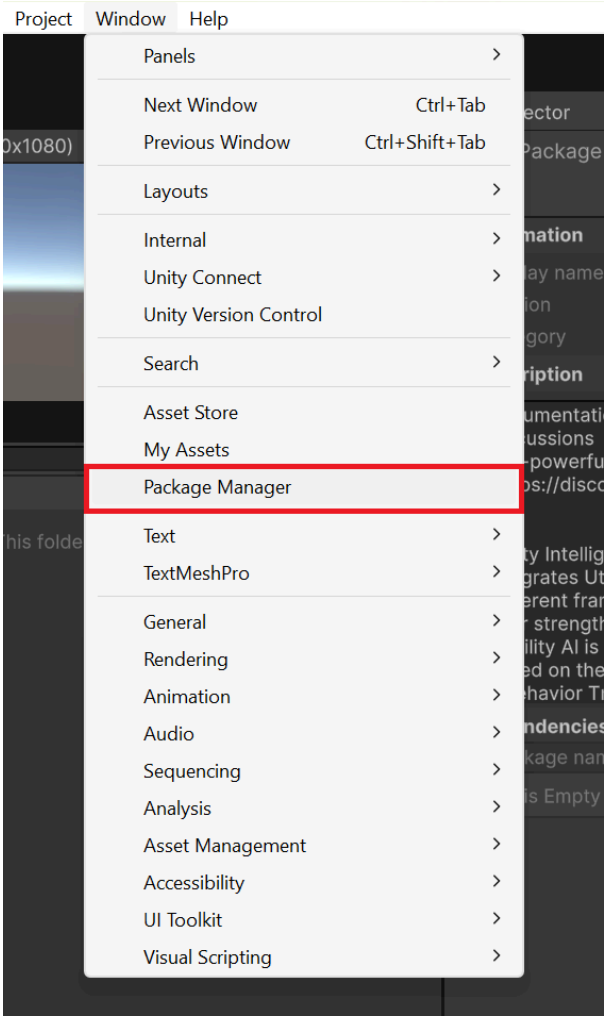
Getting Started

Installation

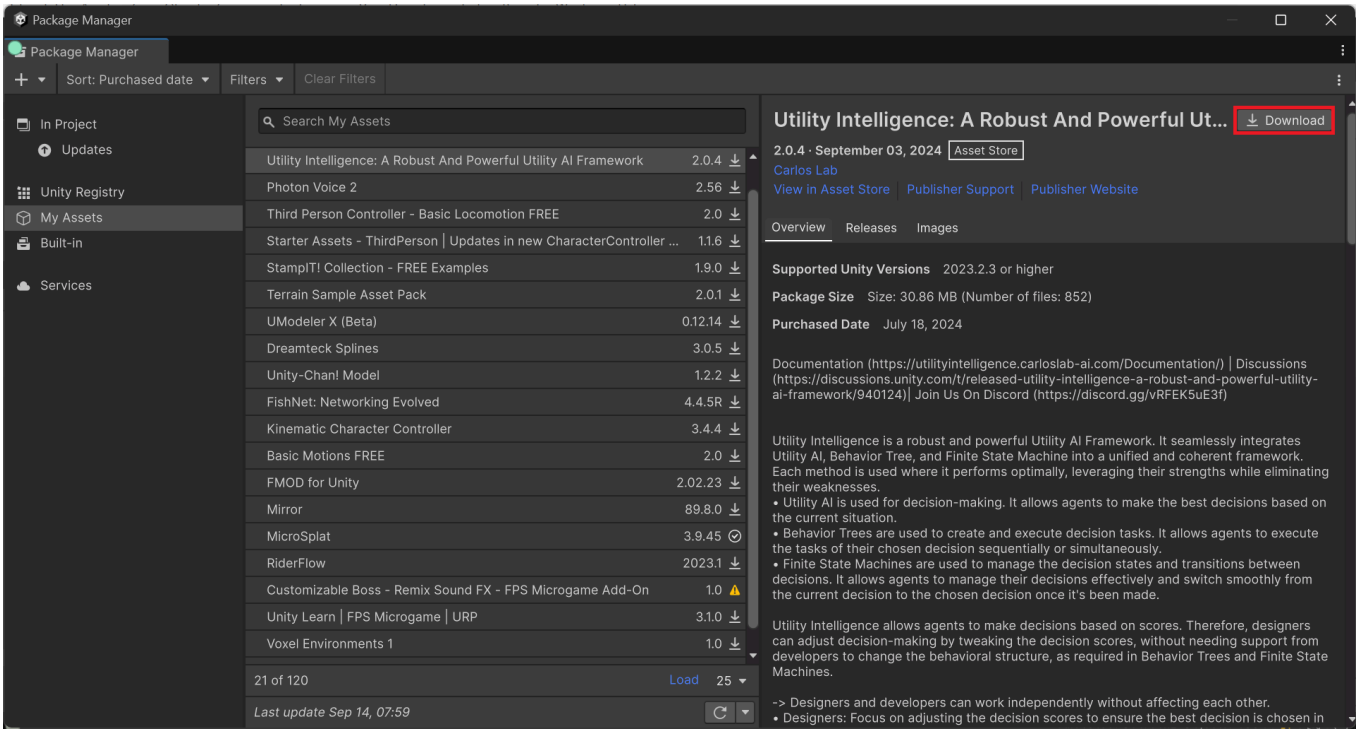
1. From **Unity Hub**, sign in to the Unity account that you used to purchase **Utility Intelligence**.



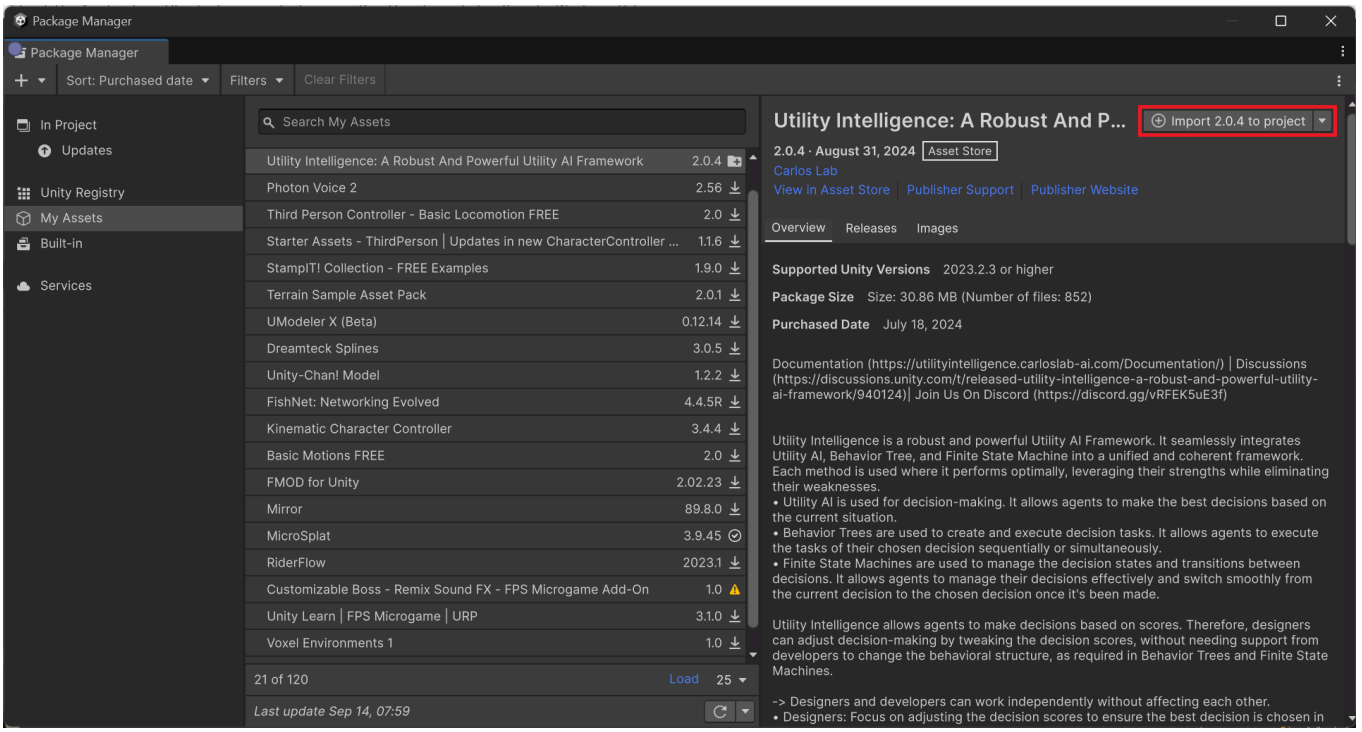
2. Open your Unity project.
3. Open the **Package Manager**.



4. Select **My Assets** -> **Utility Intelligence: A Robust And Powerful Utility AI Framework** -> **Download**



5. Click on **Import to Project**

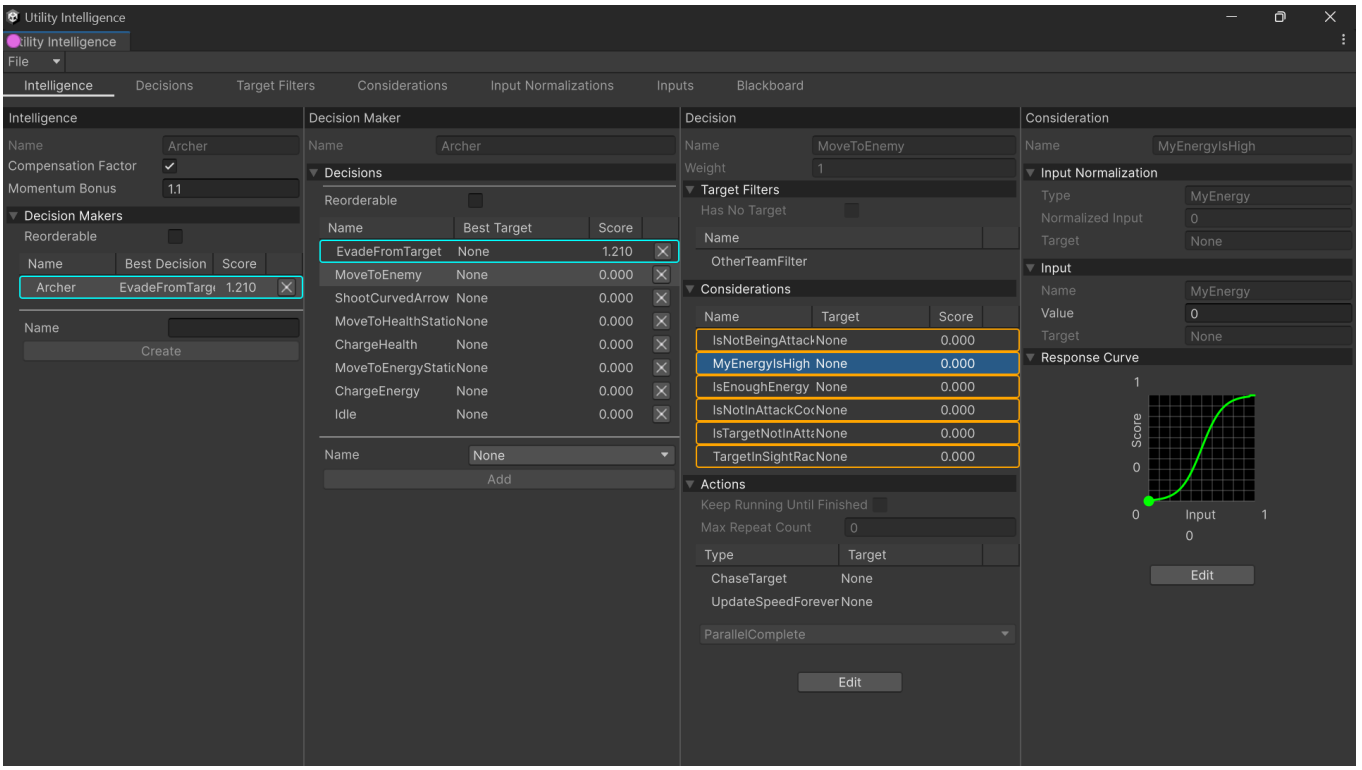


6. Enjoy exploring **Utility Intelligence** to develop your game AIs.

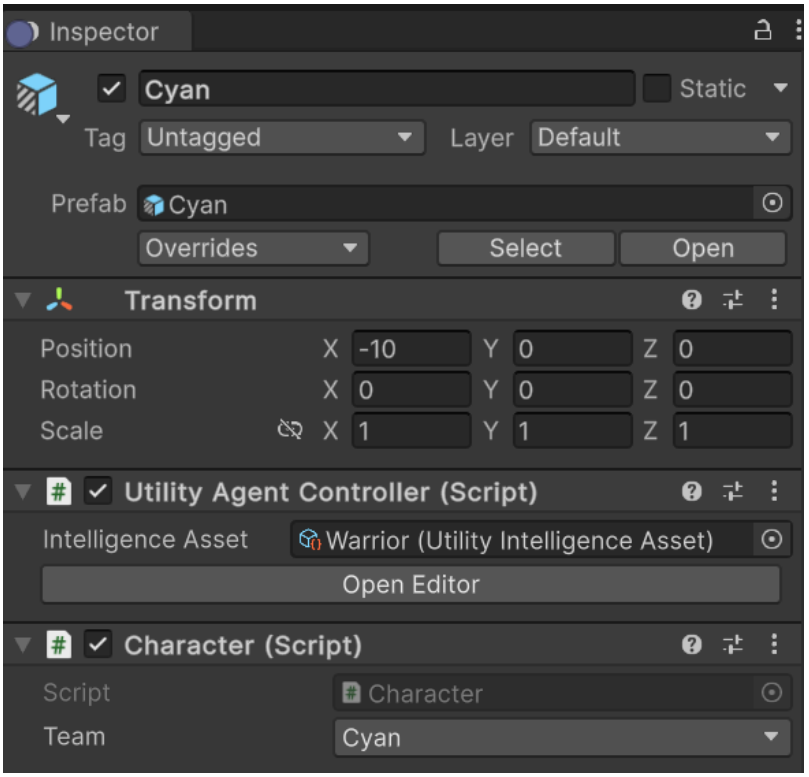
Quick Start

1. Firstly, you need to create a Utility Intelligence Asset by right-clicking in the **Project Window** and select **Create/CarlosLab/Utility Intelligence Asset**.
2. Then double-click on the new **Utility Intelligence Asset** to open the Utility Intelligence Editor.

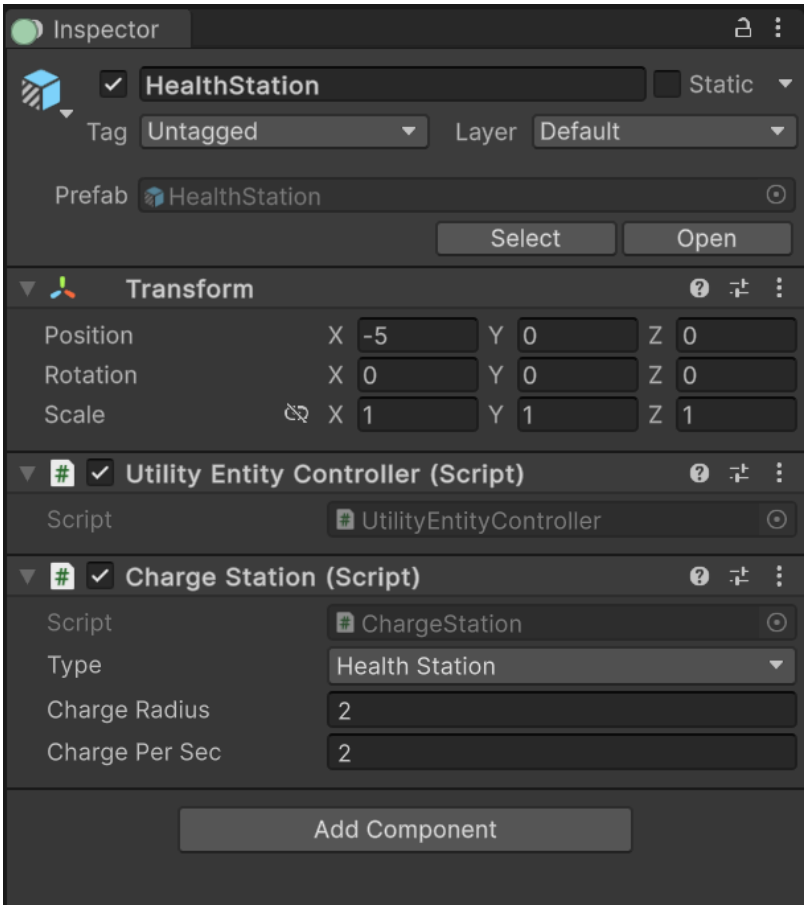
3. Add new Decision Makers, Decisions, Considerations to the intelligence asset.



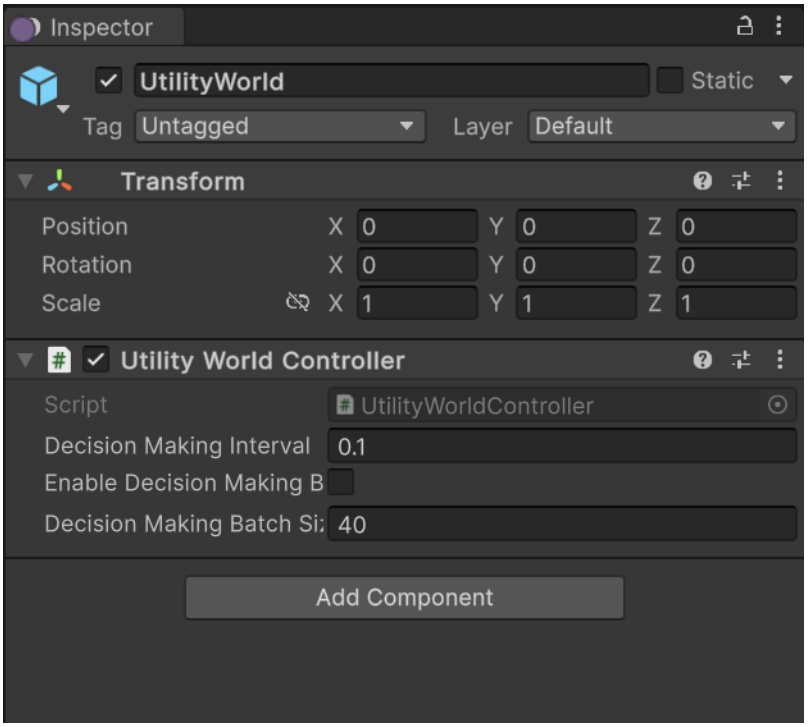
4. Transform your AI GameObjects into Utility Agents and assign the **Utility Intelligence Asset** to the **Intelligence Asset** field of the **Utility Agent Controller**



5. Transform all the Game Objects that your agents need to interact with into Utility Entities



6. Create a Utility World and register all the Utility Agents and Utility Entities in your game with it.



7. Play your game.

Example Scenes

For more information about the example scenes of this package, please visit: Example Scenes.

Other Learning Resources

Besides our documentation, there are other good learning resoures for Utility AI. You can learn a lot from them.

Texts

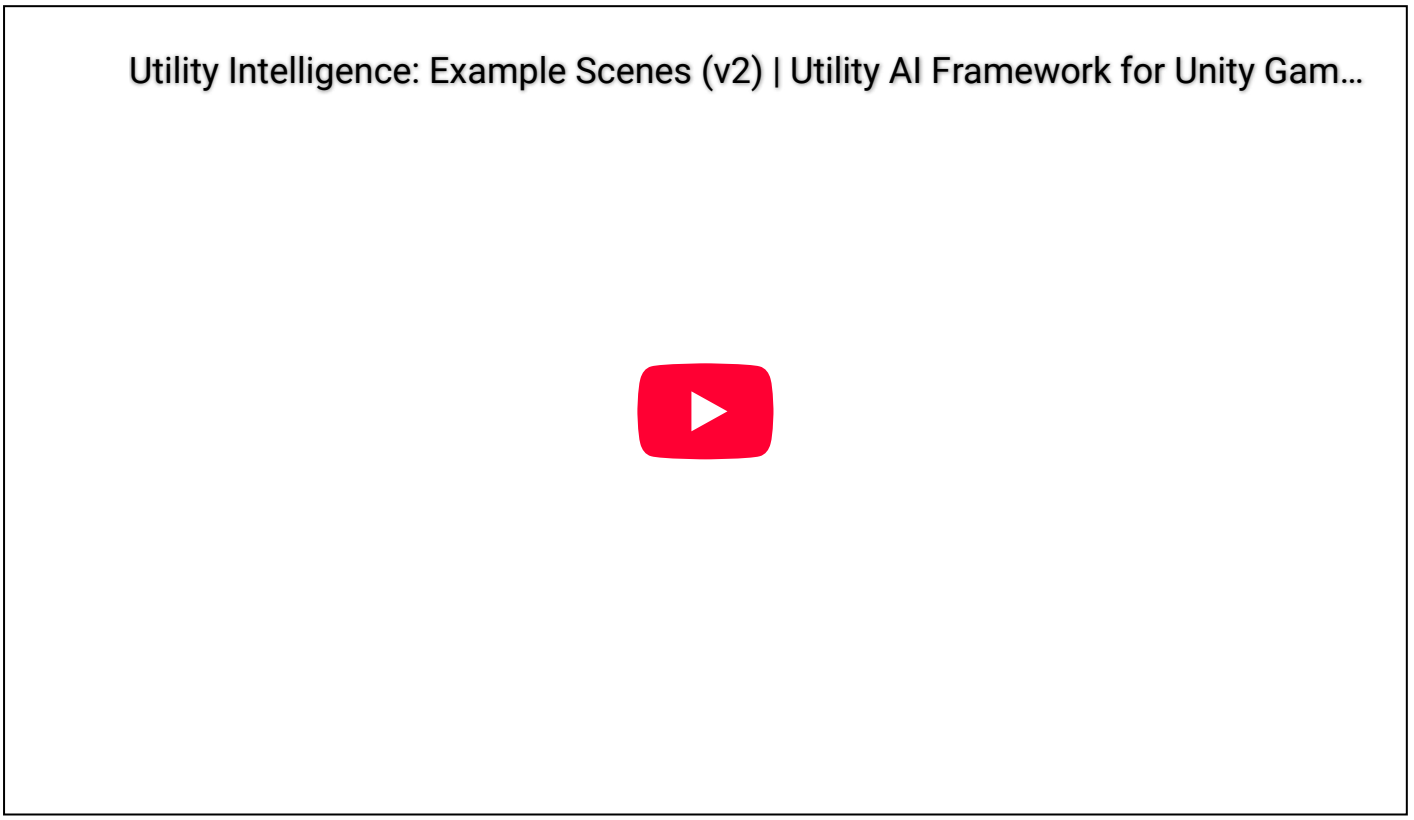
1. An Introduction to Utility Theory, David “Rez” Graham
2. Choosing Effective Utility-Based Considerations, Mike Lewis
3. Curvature’s Wiki, Mike Lewis

Videos

1. Architecture Tricks: Managing Behaviors in Time, Space, and Depth, Dave Mark (From 33:30)
2. Building a Better Centaur: AI at Massive Scale, Dave Mark and Mike Lewis

Example Scenes

We provide many example scenes to demonstrate how to use **Utility Intelligence** to create your own agents. However, by default, these examples are not imported into your project to keep it clean. If you want to learn more about **Utility Intelligence** through our examples, you need to import them into your project first.

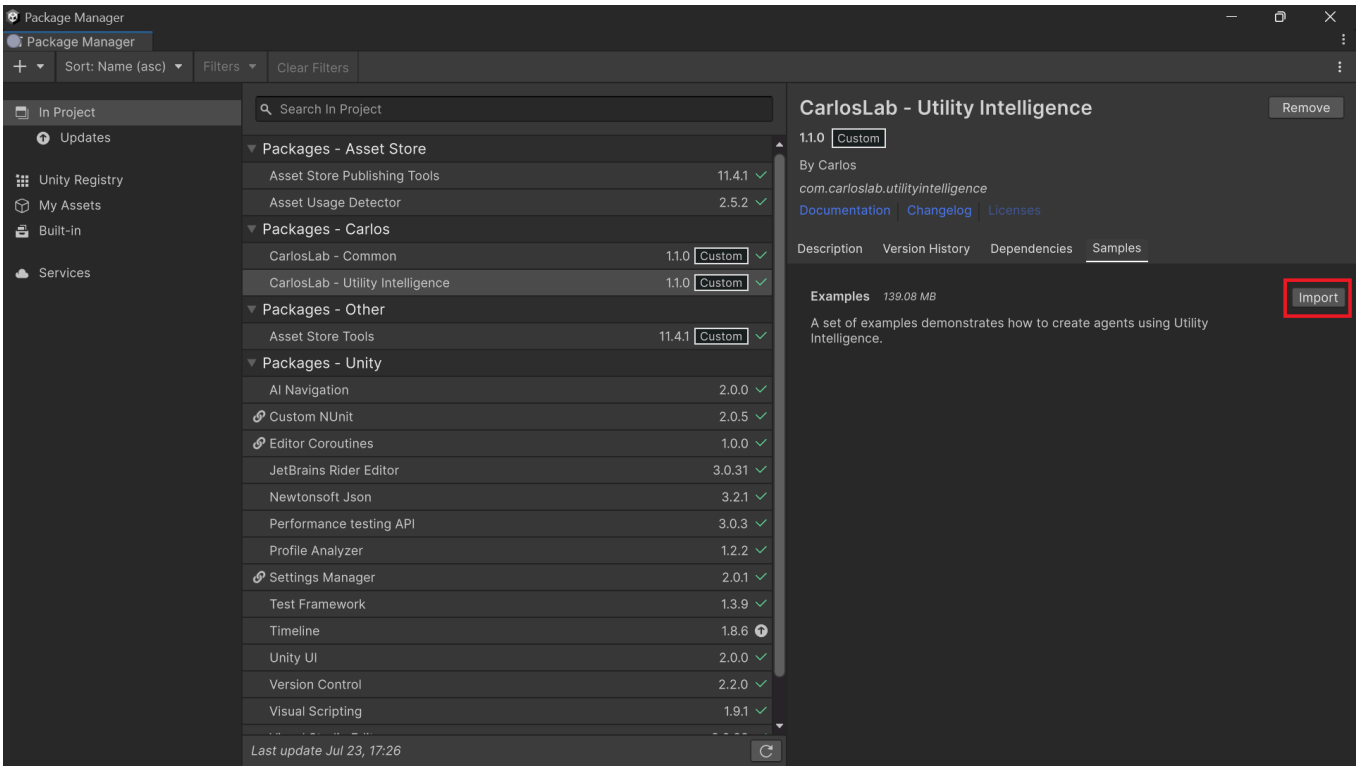


Importing example scenes

To import our example scenes to your project:

1. Open the **Package Manager**.
2. Select **In Project** -> **Carlos Lab - Utility Intelligence**.
3. Go to the **Samples** tab.

4. Click the **Import** button.



Running examples in URP and HDRP

Since this plugin doesn't have any graphical features, it is compatible with all render pipelines. However the materials of the examples are created using the Built-In Render Pipeline. Therefore, if you want to run the examples in URP or HDRP, you need to convert all materials to the target pipeline first:

URP

1. Open **Render Pipeline Converter** (Window -> Rendering -> Render Pipeline Converter).
2. Tick **Material Upgrade**.
3. Click **Initialize and Converter** button.

Or

1. Select all materials in our examples.
2. Click **Edit -> Rendering -> Material -> Convert Selected Built-in Materials to URP**.

HDRP

1. Open **HDRP Wizard** (Window -> Rendering -> HDRP Wizard).
2. Click **Convert All Built-In Materials to HDRP**.

Or

1. Select all materials in our examples.
2. Click **Edit -> Rendering -> Material -> Convert Selected Built-in Materials to HDRP**.

Utility World

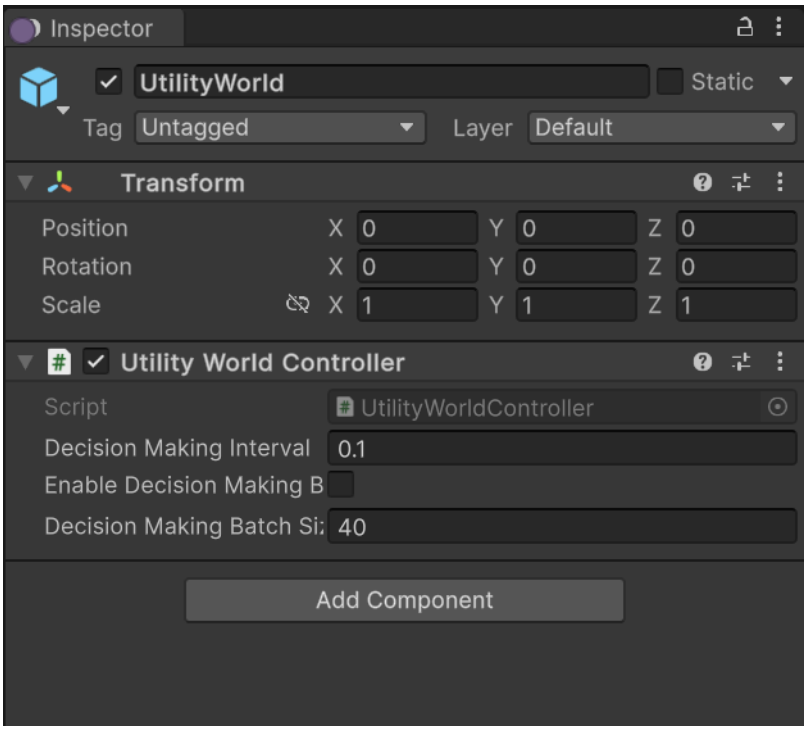
Utility World

A **Utility World** contains a collection of Utility Entities, and
the main roles of a Utility World are:

1. Handling the decision-making process of all Utility Agents inside the world.
2. Running the task associated with the chosen decision for all Utility Agents inside the world.

> **Note**
>
> - Utility Worlds manage their Utility Entities and Utility Agents independently, not related to each other.
> - So you can create multiple utility worlds for different purposes without having to worry about they will affect each other.

To create a Utility World, right-click in the **Hierarchy Window**, then select **CarlosLab/Utility World**. Alternatively, you can create it manually by creating a new Game Object and adding a **Utility World Controller** component to it:



The **Utility World Controller** will automatically create a **Utility World** when your game starts and manage it throughout its lifetime.

> **Tip**
>
> You can optimize the decision-making process of each **Utility World** by adjusting the Decision Making Interval and the Decision Making Batch Size.

Utility Entity

A Utility Entity represents an object inside a Utility World, and only Utility Entities in the same world can interact with each other. Therefore, if you want a GameObject to be the target of a Utility Agent, you need to do the following:

1. Transform the GameObject into a Utility Entity
2. Register the Utility Entity with the same Utility World as the Utility Agent.

Transforming GameObjects into Utility Entities

To transform a GameObject into a Utility Entity, you need to attach these two components to it:

1. **Utility Entity Facade**

- It is used to interact with the **Utility Entity**'s Game Object. For example, Target Filters can access the **Entity Facade** of both itself and the target to retrieve information from the components of Game Objects in order to check the validity of the target.

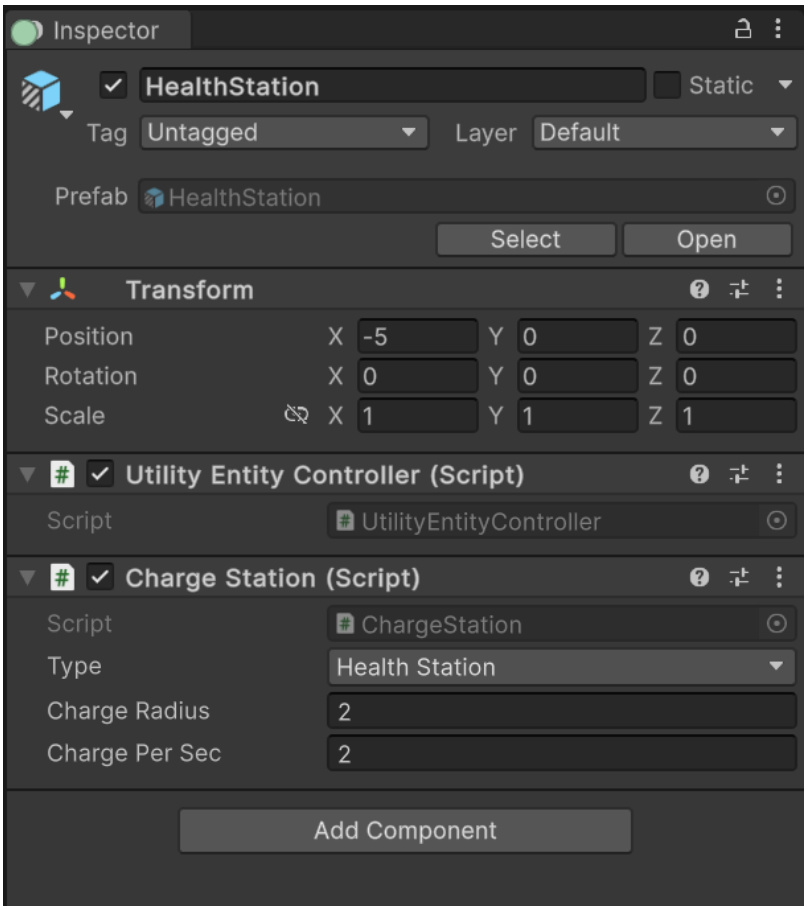
```
public class OtherTeamFilter : TargetFilter
{
    protected override bool OnFilterTarget(UtilityEntity target)
    {
        if (target.EntityFacade is Character targetCharacter)
        {
            Character myCharacter = AgentFacade as Character;
            return myCharacter.Team != targetCharacter.Team;
        }

        return false;
    }
}
```

- To create your own **Entity Facade**, you need to create a class inherited from `UtilityEntityFacade`. For example:

```
public class ChargeStation : UtilityEntityFacade
{
    [SerializeField]
    private ChargeStationType type;

    [SerializeField]
    private float chargeRadius;

    [SerializeField]
    private float chargePerSec;

    public ChargeStationType Type => type;
    public float ChargeRadius => chargeRadius;
    public float ChargePerSec => chargePerSec;
}
```

2. **Utility Entity Controller**

- The main role of a Utility Entity Controller is to create and manage the Utility Entity's lifecycle, including initialization, destruction, registration, and unregistration with utility worlds.



Registering Utility Entities

> **Note**
>
> - A Utility Entity can only be associated with a single Utility World.
> - Therefore, it's not possible to register a Utility Entity with multiple Utility Worlds.

To register a Utility Entity with a Utility World, you need to call the `Register` method of the **UtilityEntityController** and pass the Utility World as the parameter. For example:

```
public class AgentsPlacedInSceneDemo : MonoBehaviour
{
    [SerializeField]
    private UtilityWorldController world;

    [SerializeField]
    private List<UtilityAgentController> agents;

    [SerializeField]
    private List<UtilityEntityController> chargeStations;

    private void Start()
    {
        foreach (UtilityAgentController agent in agents)
        {
            agent.Register(world);
        }

        foreach (UtilityEntityController chargeStation in chargeStations)
        {
            chargeStation.Register(world);
        }
    }
}
```

Getting Utility Entities

After being registered with a Utility World, the Utility Entity is allocated an **Entity Id**. This Id is unique within the world, and you can get the entity from the world by calling `UtilityWorldController.GetEntity()` and passing the **Entity Id** as the parameter of the method. For example:

```
int entityId = entity.Id;
var entity = world.GetEntity(entityId);
```

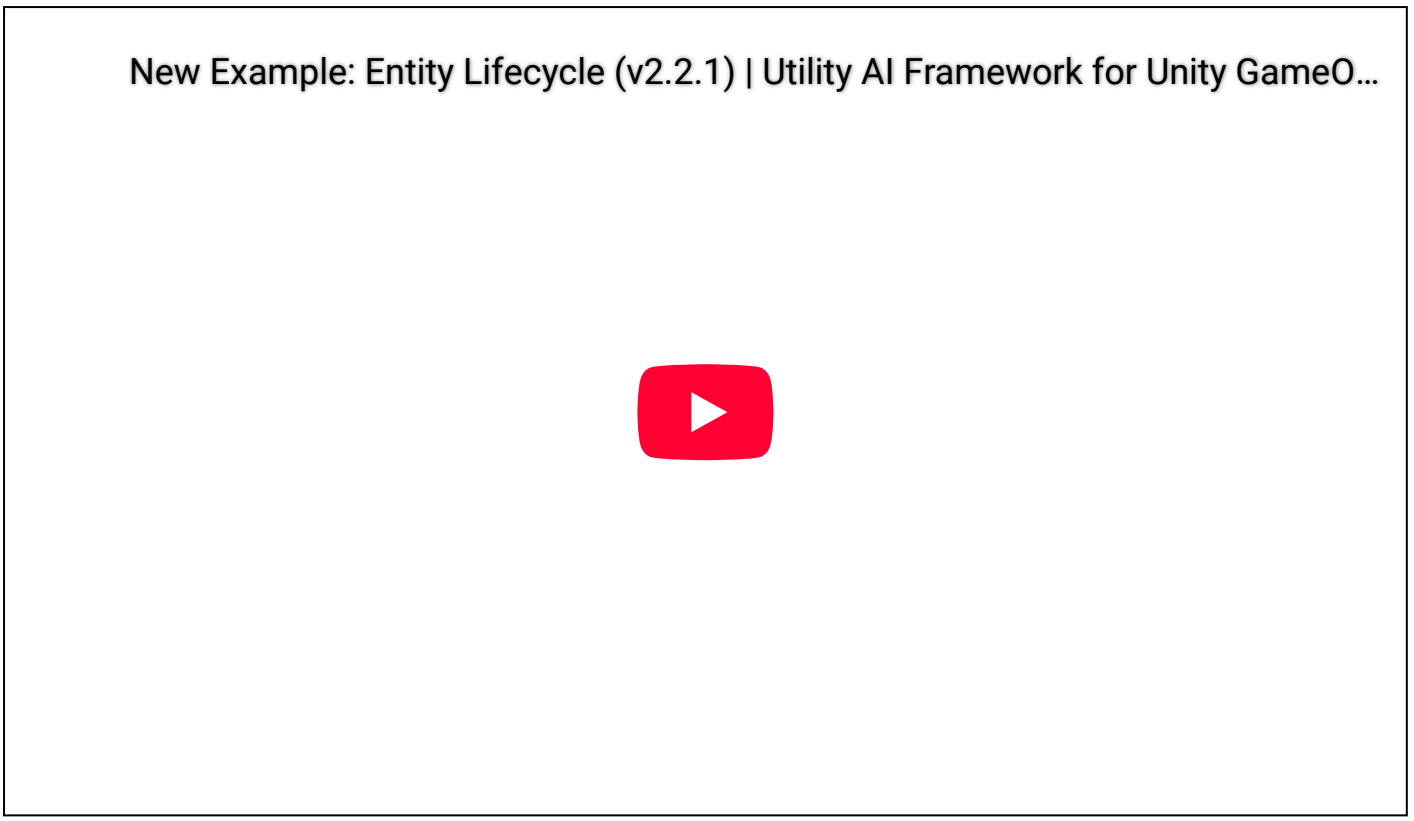
It's useful in case you want to access the entity from multiple places but don't want to pass the entity object everywhere.

Entity Lifecycle

In **v2.2.1**, I added these lifecycle event functions to `EntityFacade`. You can override these functions to receive notifications when lifecycle events occur.

```
protected virtual void OnRegistered()
{
}

protected virtual void OnActivated()
{
}

protected virtual void OnEnabled()
{
}

protected virtual void OnDisabled()
{
}

protected virtual void OnDeactivated()
{
}

protected virtual void OnUnregistered()
{
}

protected virtual void OnDestroyed()
{
}
```

Additionally, **v2.2.1** includes a new example to demonstrate the lifecycle of utility entities:



Since utility entities are managed by a utility world, performing the following actions within action tasks is unsafe because they directly affect the utility world, which is also responsible for running action tasks:

- Register/Unregister utility entities.
- Activate/Deactivate utility entities.
- Enable/Disable utility entities.
- Destroy utility entities.

For safety, you should use these functions inside action tasks instead. They will be queued to run after all action tasks have executed.

- `EntityController.Register()`
- `EntityController.Unregister()`
- `EntityController.SetActive()`
- `EntityController.Activate()`
- `EntityController.Deactivate()`
- `EntityController.SetEnable()`
- `EntityController.Enable()`
- `EntityController.Disable()`
- `EntityController.Destroy()`

Or:

- `EntityFacade.Register()`
- `EntityFacade.Unregister()`
- `EntityFacade.SetActive()`
- `EntityFacade.Activate()`
- `EntityFacade.Deactivate()`
- `EntityFacade.SetEnable()`
- `EntityFacade.Enable()`
- `EntityFacade.Disable()`
- `EntityFacade.Destroy()`

If it is outside of action tasks, you can use these functions instead. They will be run immediately without queueing.

- `EntityController.RegisterImmediate()`
- `EntityController.UnregisterImmediate()`
- `EntityController.SetEnableImmediate()`
- `EntityController.EnableImmediate()`
- `EntityController.DisableImmediate()`

Or:

- `EntityFacade.RegisterImmediate()`
- `EntityFacade.UnregisterImmediate()`
- `EntityFacade.SetEnableImmediate()`
- `EntityFacade.EnableImmediate()`
- `EntityFacade.DisableImmediate()`

And:

- `GameObject.SetActive`
- `GameObject.Destroy`

Utility Agent

A Utility Agent is a special Utility Entity that helps your AI make the right decision based on the current situation, and controls it to perform the Action Tasks attached to the chosen decision.

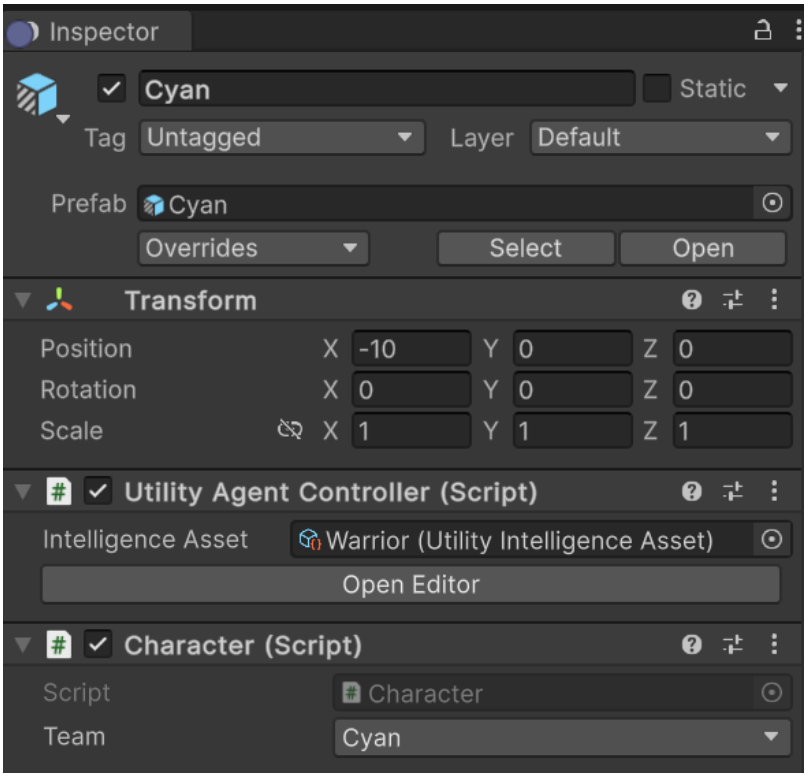
Transforming GameObjects into Utility Agents

To transform a Game Object into a Utility Agent, you need to attach these two components to it:

1. **Utility Agent Facade**
   - It is similar to **Utility Entity Facade** but instead of interact with the GameObject of the **Utility Entity**, it is used to interact with the Game Object of the **Utility Agent**.
   - To create your own Utility Agent Facade, you need to create a class inherited from `UtilityAgentFacade`. For example:

```
public class Character : UtilityAgentFacade
{
    [SerializeField]
    private Team team;

    private CharacterEnergy energy;
    private CharacterHealth health;
    private NavMeshAgent navMeshAgent;
    private Rigidbody rigidBody;

    public Team Team => team;
    public NavMeshAgent NavMeshAgent => navMeshAgent;
    public Rigidbody RigidBody => rigidBody;
    public CharacterHealth Health => health;
    public CharacterEnergy Energy => energy;

    private void Awake()
    {
        navMeshAgent = GetComponent<NavMeshAgent>();
        rigidBody = GetComponent<Rigidbody>();

        health = GetComponent<CharacterHealth>();
        energy = GetComponent<CharacterEnergy>();
    }
}
```

2. **Utility Agent Controller**
   - It is similar to **Utility Entity Controller**, but instead of create and manage the lifecycle of the **Utility Entity**, it creates and manage the lifecycle of the **Utility Agent**.
   - It injects Utility Intelligence Data from the **Utility Intelligence Asset** into Utility Agent, giving the agent intelligence.


Inspector
Cyan
Static
Tag
Untagged
Layer
Default
Prefab
Cyan
Overrides
Select
Open
Transform
Position
X -10
Y 0
Z 0
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Utility Agent Controller (Script)
Intelligence Asset
Warrior (Utility Intelligence Asset)
Open Editor
Character (Script)
Script
Character
Team
Cyan

Utility Intelligence

Utility Intelligence

Utility Intelligence is an object that uses Utility Intelligence Data to help Utility Agents make and execute decisions. It grants intelligence to **Utility Agents**.

Utility Intelligence Asset

Utility Intelligence Asset is a data container used to store Utility Intelligence Data. It can be created by right-clicking in the **Project Window** and select **Create/CarlosLab/Utility Intelligence Asset**.

Utility Intelligence Data

Utility Intelligence Data is stored in **JSON** format. It includes information about:

- Decision Makers
- Decisions
- Target Filters
- Considerations

There are two ways to edit **Utility Intelligence Data**:

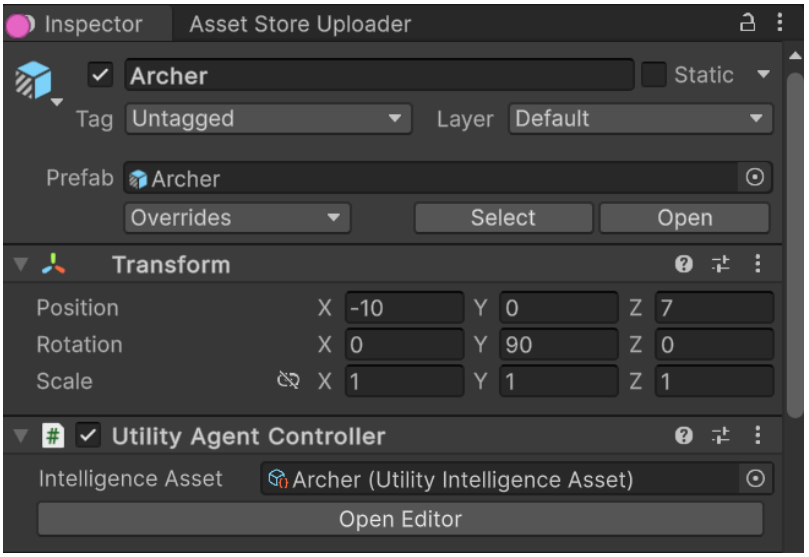
1. **Manually Editing**: Use a text editor to edit the data, and then import it into **Utility Intelligence Asset** using File Toolbar Menu.
2. **Utility Intelligence Editor**: Use the Utility Intelligence Editor to edit the data. This is the recommended way.

Intelligence Editor

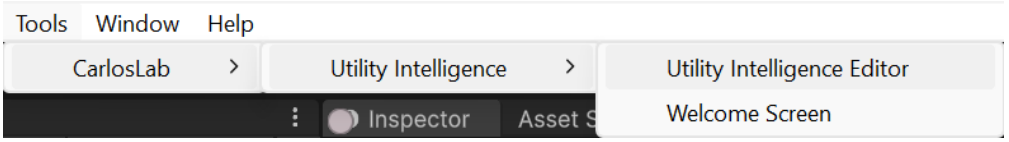
Editor Mode

There are three ways to open the **Utility Intelligence Editor** for a Utility Intelligence Asset at editor time (**Editor Mode**):

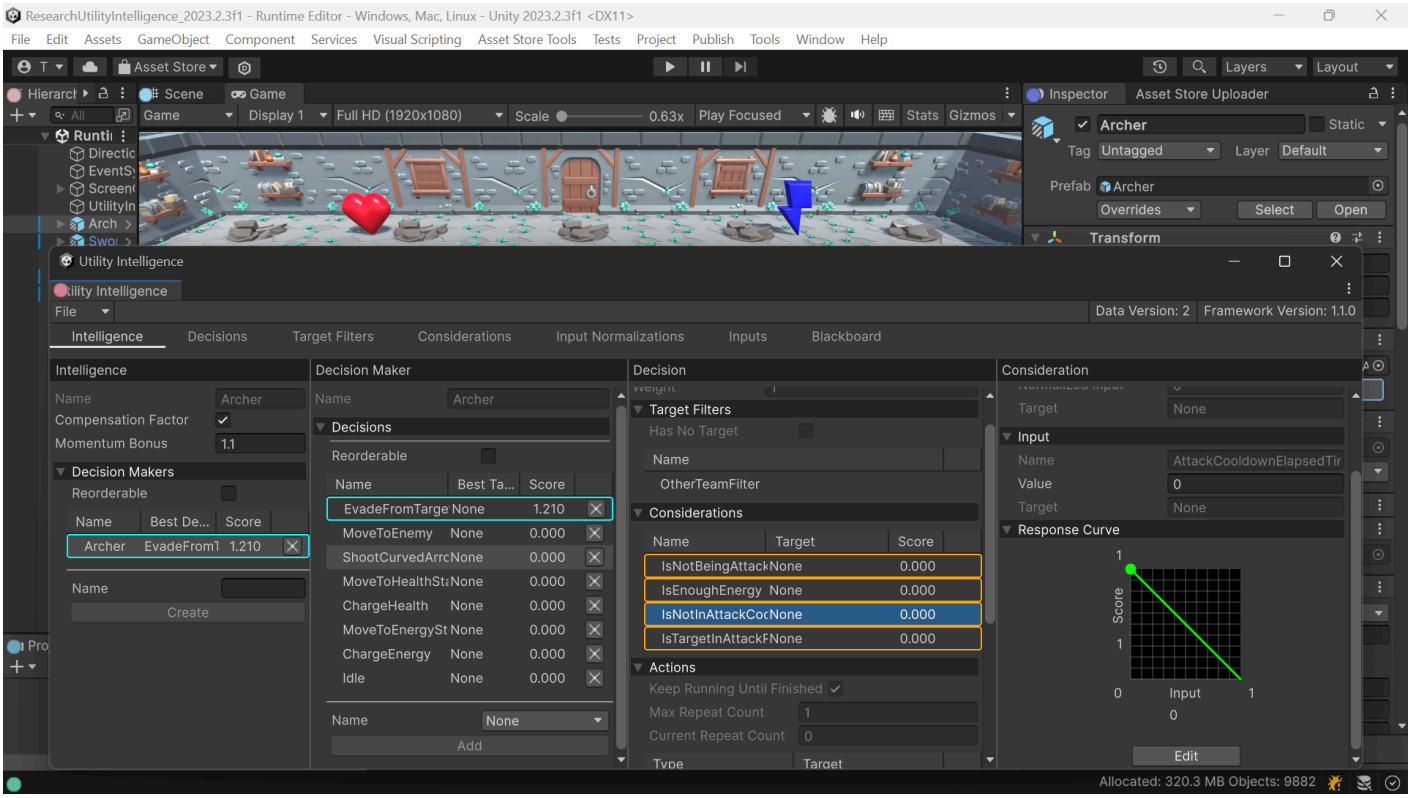
1. Double-click the **Utility Intelligence Asset** in the Project Window.
2. Select the **Utility Agent** with the assigned **Utility Intelligence Asset** in the Hierarchy Window, and then click the **Open Editor** button in the Inspector Window.



3. Select **Tools -> Carlos Lab -> Utility Intelligence -> Utility Intelligence Editor**, and then select the **Utility Intelligence Asset** in the Project Window.



Here's how the **Utility Intelligence Editor** looks in **Editor Mode**:



Toolbar

Currently, Unity only supports toolbar in Editor Mode, so you won't see it if you open it in Runtime Mode.

File Menu

- `Import Data`: Import the **Intelligence Data** from a JSON file.
- `Export Data`: Export the **Intelligence Data** to a JSON file.
- `Show Data`: Show the **Intelligence Data** in JSON format.
- `Clear Data`: Clear all the **Intelligence Data**.

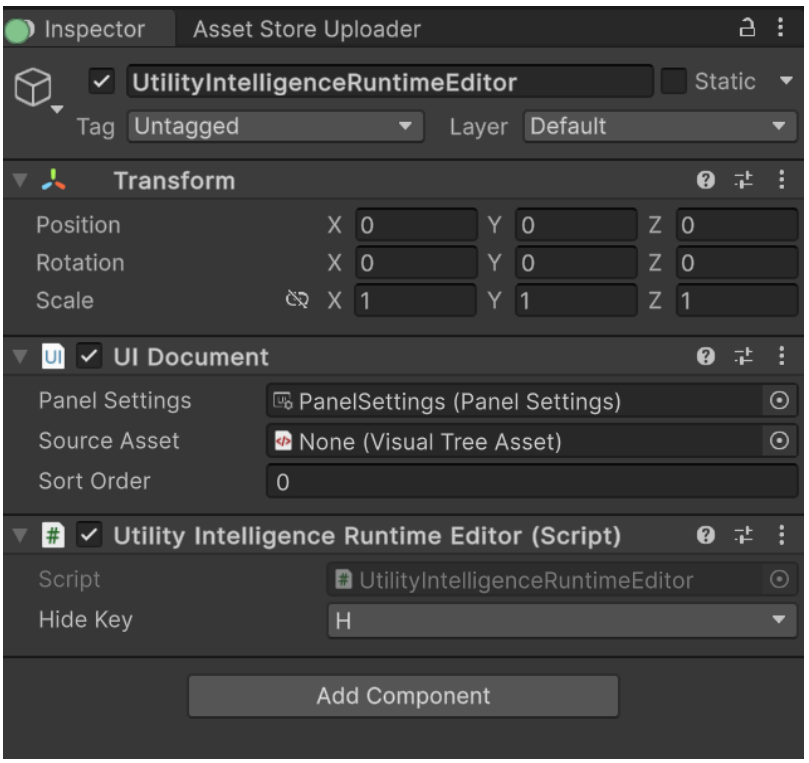
With the File Menu Toolbar, you can edit the **Intelligence Data** directly in JSON format using your Text Editor, then import it into the **Intelligence Asset**:

Runtime Mode

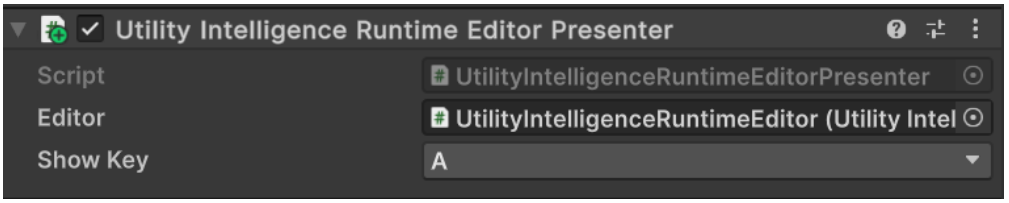
One robust feature **Utility Intelligence** offers is that the **Utility Intelligence Editor** can function both during editor time and at runtime in builds. This feature **enables users** to **adjust variables** in the **Utility Intelligence Editor** to **observe** how they **affect** the agent's **decisions** for testing purposes **in builds**.

To open the **Utility Intelligence Editor** for a Utility Agent at runtime in builds (**Runtime Mode**):

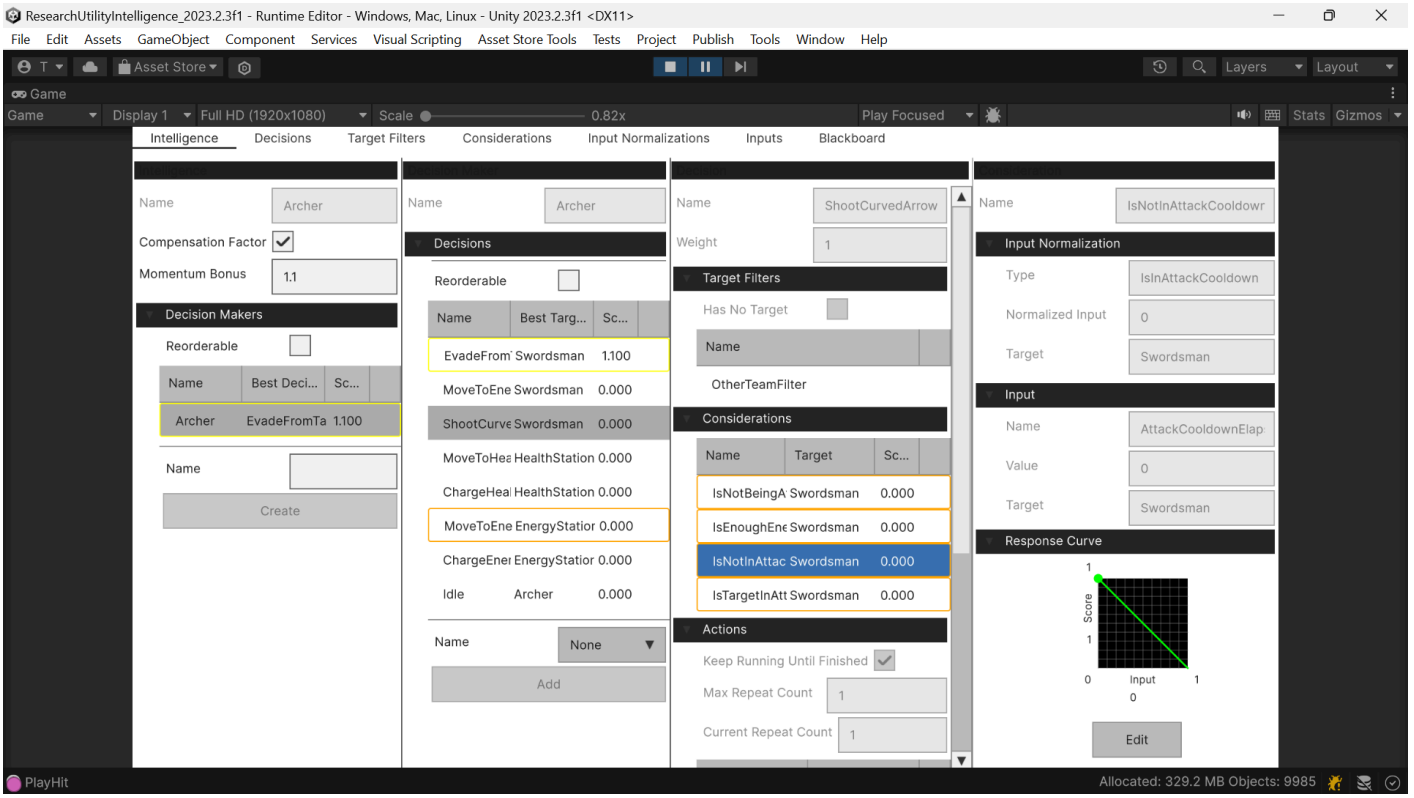
1. Create a **Utility Intelligence Runtime Editor** by right-clicking in the Hierarchy Window, then select **CarlosLab/Utility Intelligence Runtime Editor**. Alternatively, you can create it manually by creating a new Game Object and adding these components to it.



2. Add a **Utility Intelligence Runtime Editor Presenter** to the **Utility Agent**, assign the **Utility Intelligence Runtime Editor** to the **Editor** field, and set the **Show Key** to show the editor when the key is pressed.



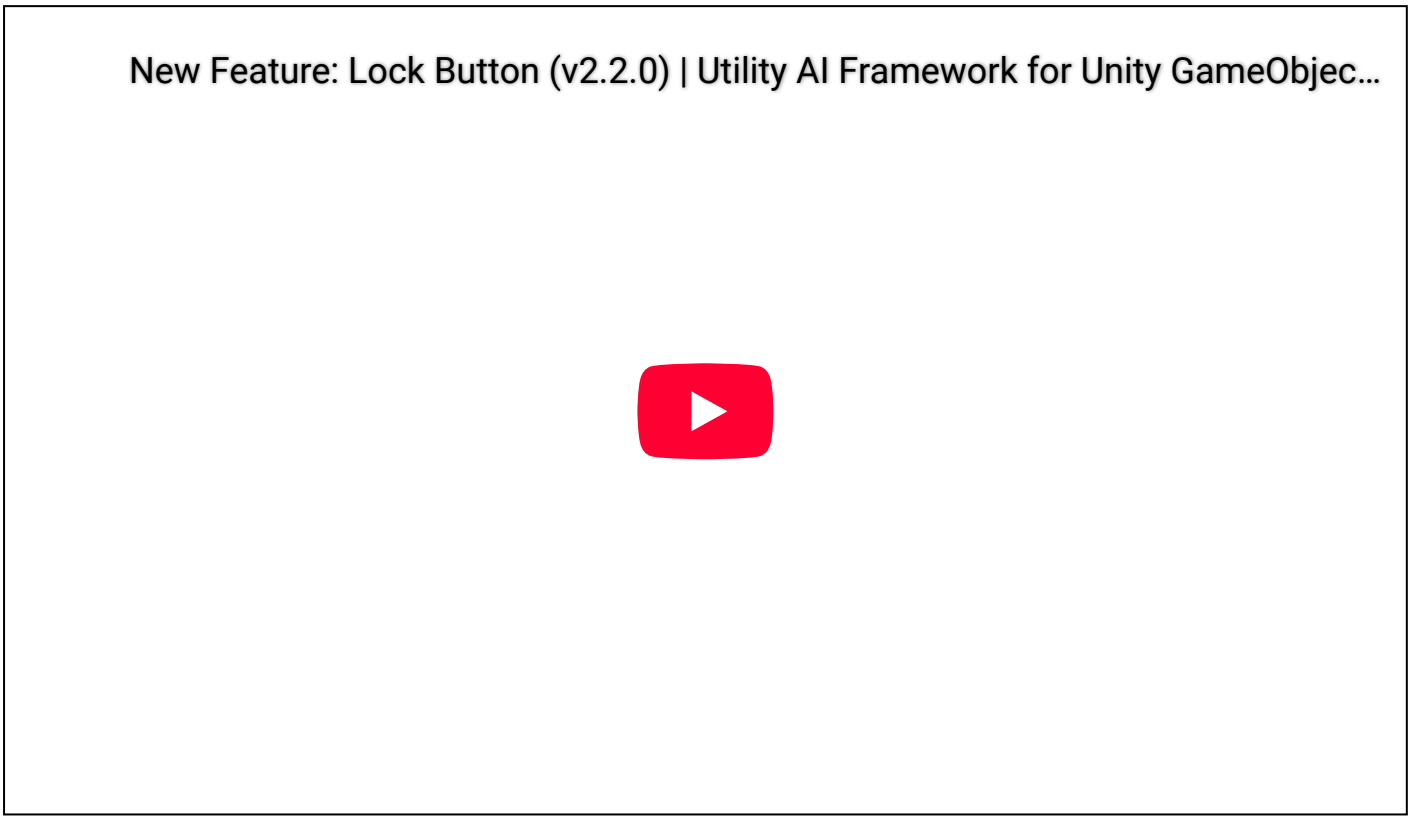
Here's how the **Utility Intelligence Editor** looks in **Runtime Mode**:



Feature: Runtime Editor (v2)

Lock the Editor

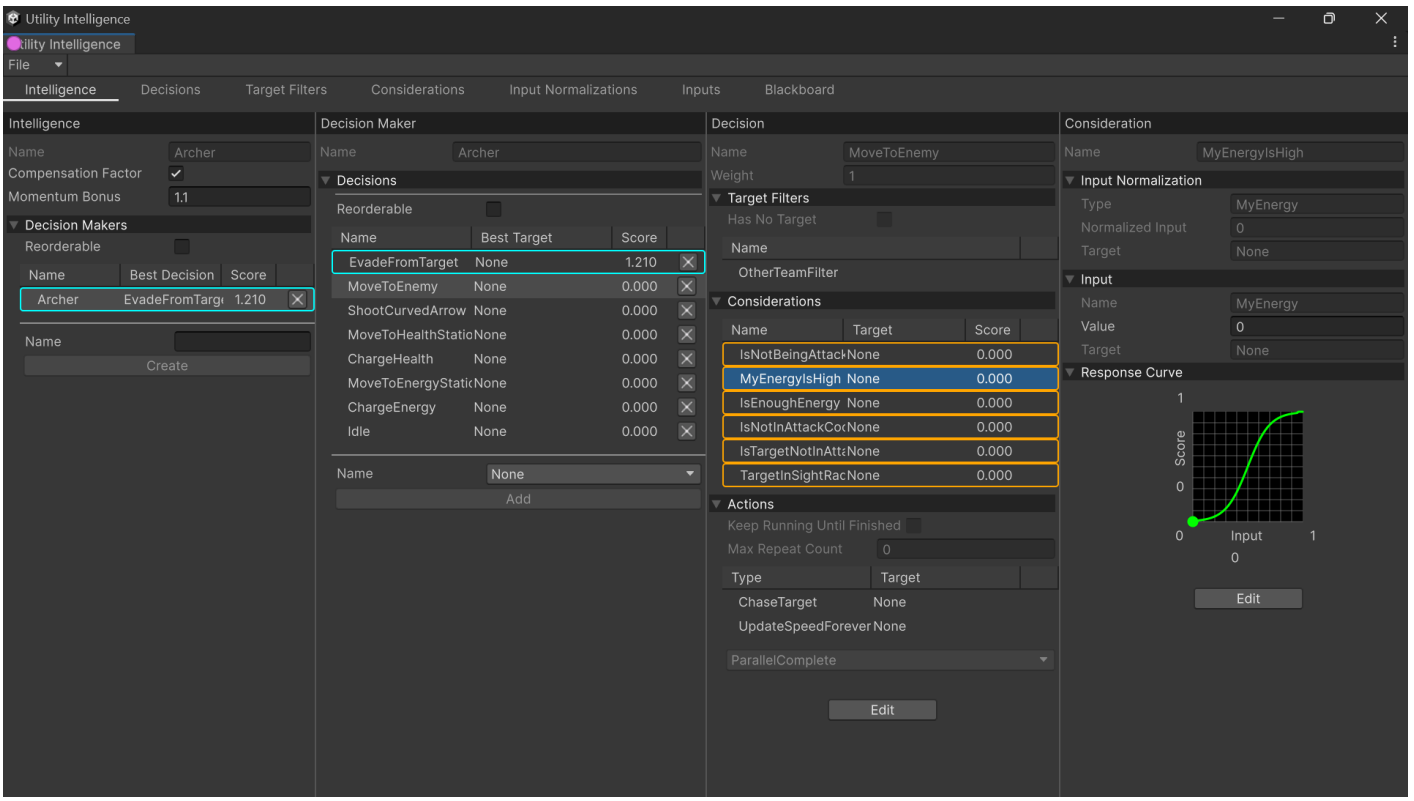
We have received feedback that the ability to lock the **Intelligence Editor** is important for testing purposes. It allows users to modify variables from other **Game Objects** through the **Inspector Window** and see how they affect the decision scores in the **Intelligence Editor**. After considering, we decided to add this feature in **v2.2.0**. We believe you will like this feature.



Tabs

Intelligence Tab

In **Intelligence Tab**, you can create new decision makers and add Decisions created in Decision Tab to your decision makers.

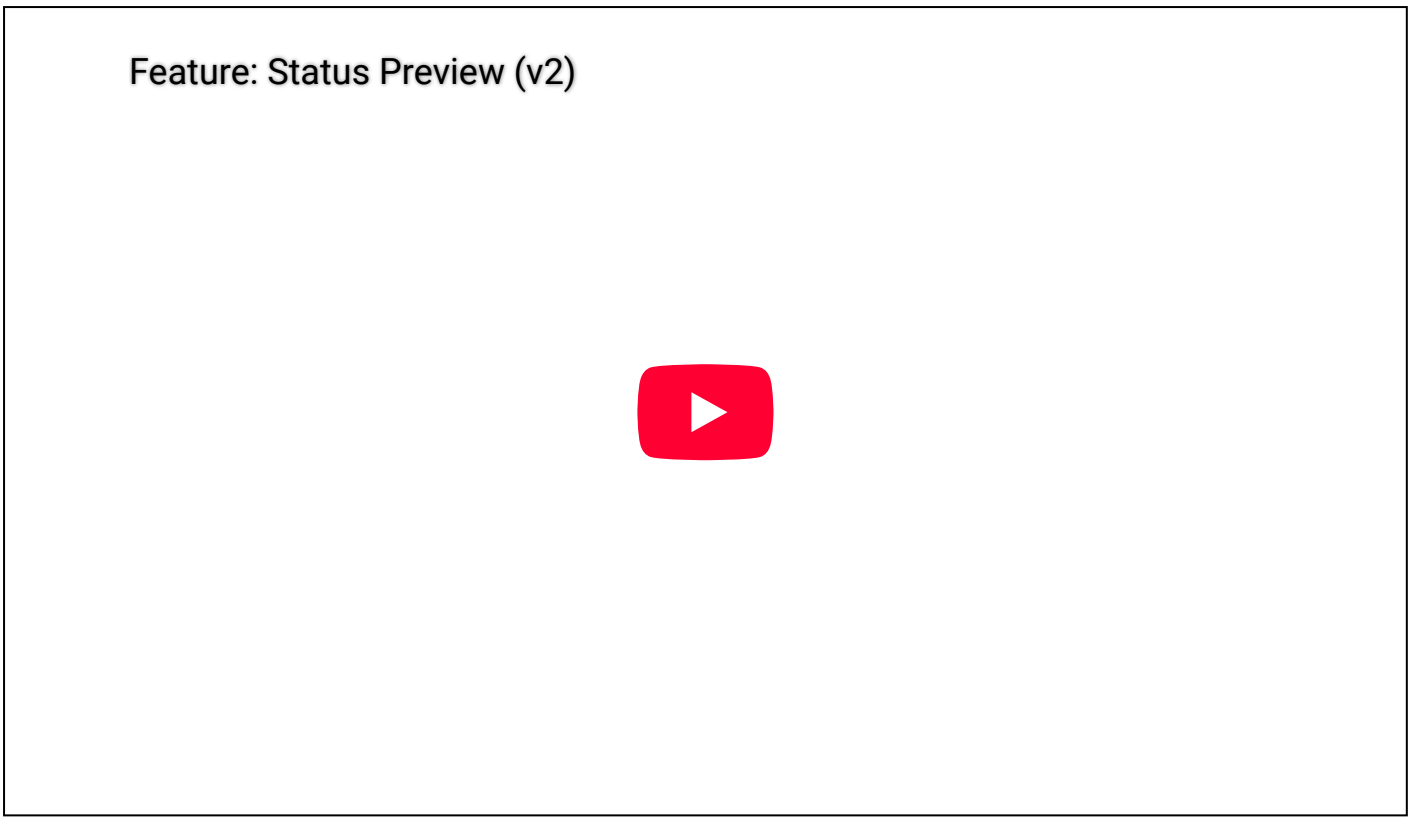


Status Preview

Besides that, you can preview the status of multiple components for any changes, such as inputs, and response curves, **right in the Editor without having to play** your game. For example:

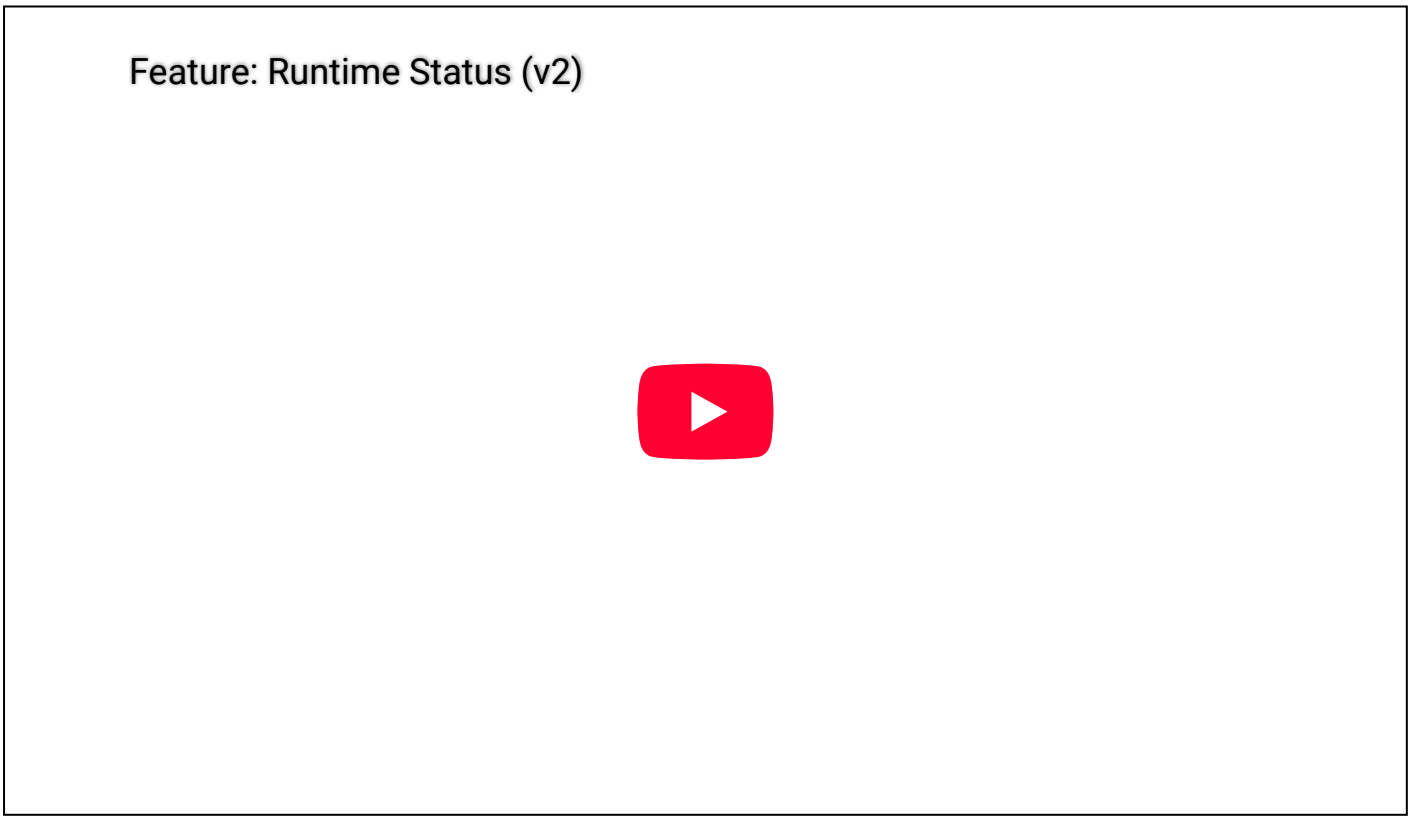
- The score and status of each consideration, indicating which considerations are executed and discarded.
- The score and status of each decision, indicating which decision is chosen based on the current inputs, input normalizations, and response curves.

I believe this feature will save a lot of your time while designing AIs for your games.



Runtime Status

Additionally, you can view the current status of multiple components during runtime. It is similar to **Status Preview** but includes additional runtime information, such as the **best target** for each decision, and the **current status** of considerations and action tasks.

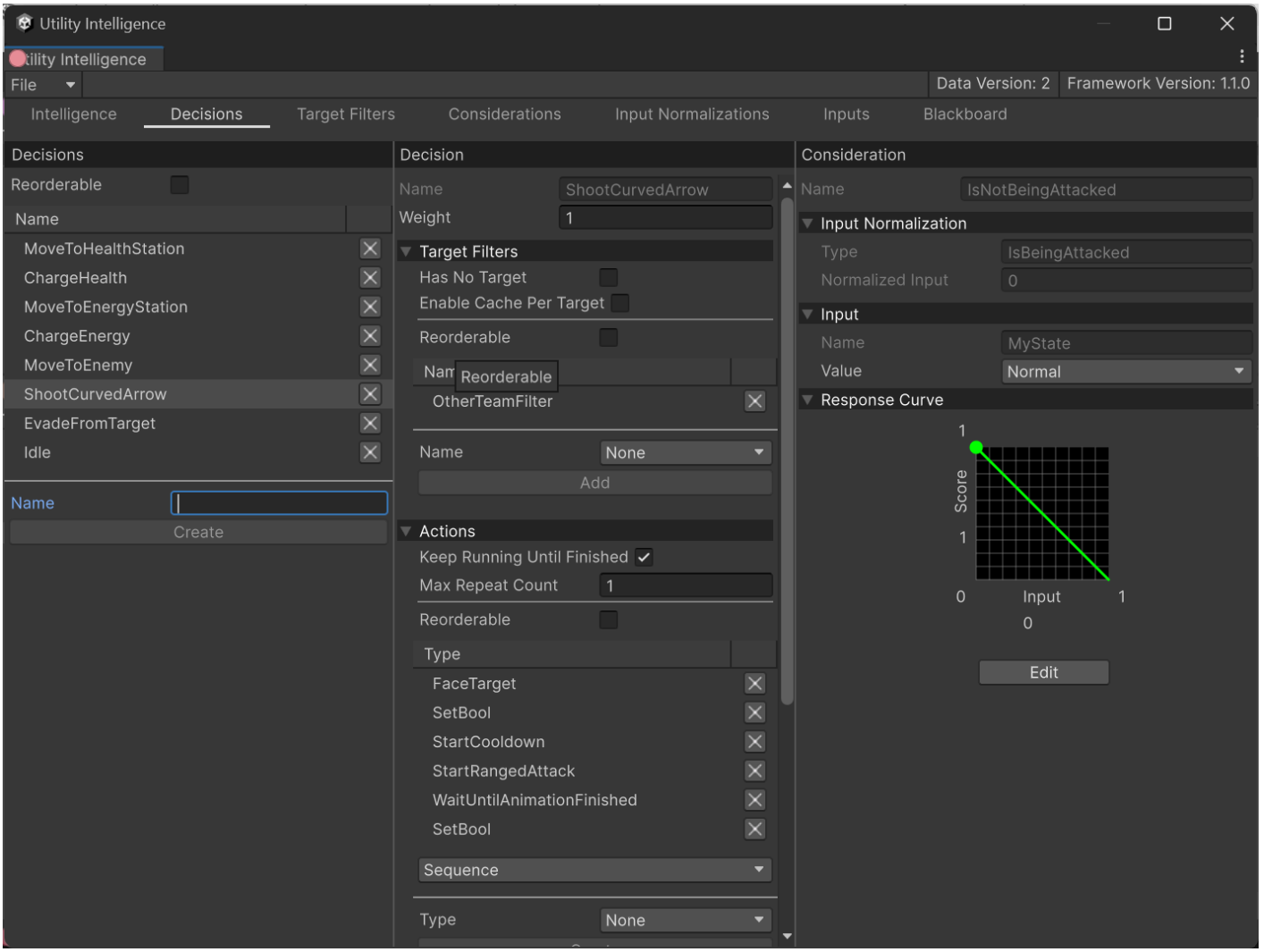


Runtime Editing

Furthermore, you can **tweak** your **AI behaviors** during **runtime** for testing purposes **without** having to **replay** your game.

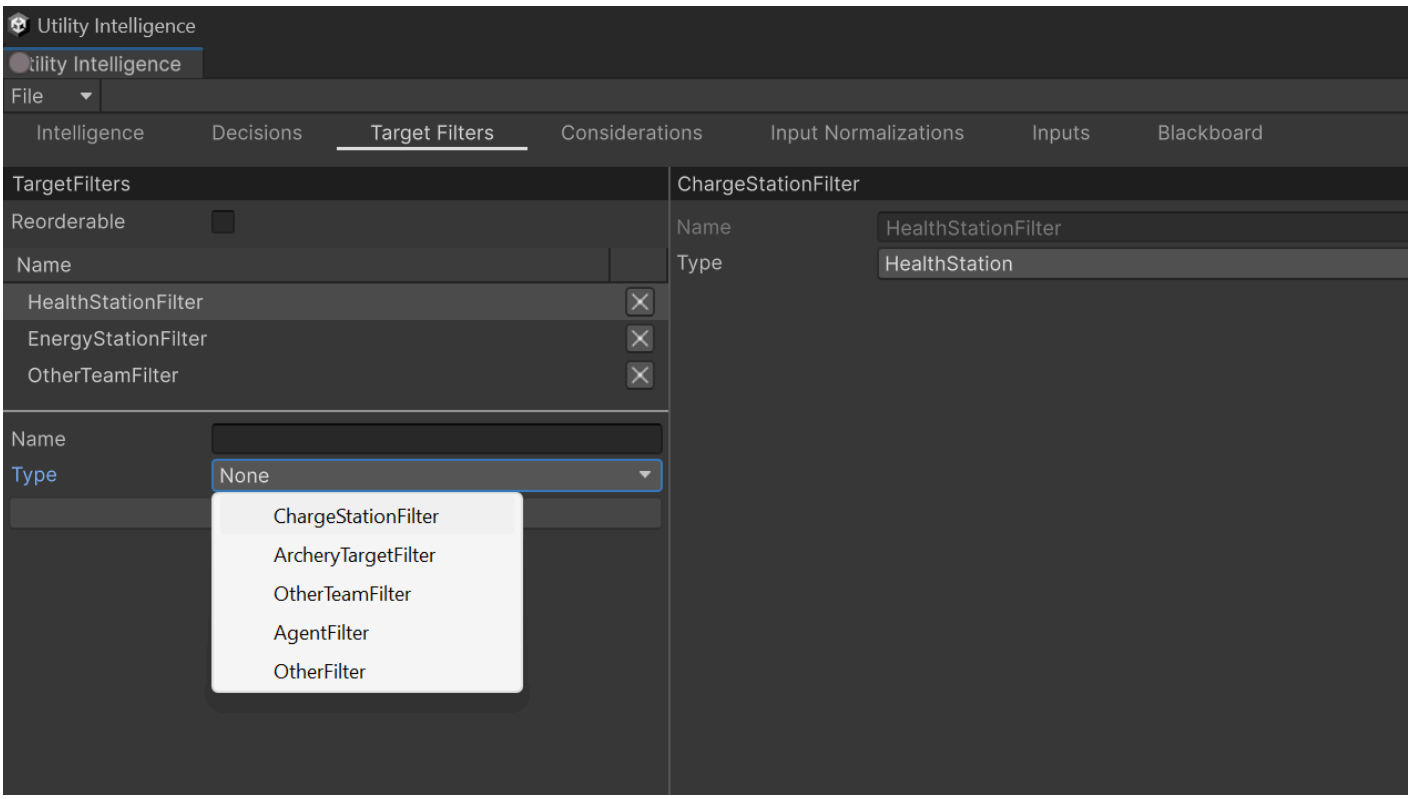
Decision Tab

In **Decision Tab**, you can create new Decisions and add target filters, action tasks, Considerations to your decisions.



Target Filter Tab

In **Target Filter Tab**, you can create new target filters to filter targets for each decision:

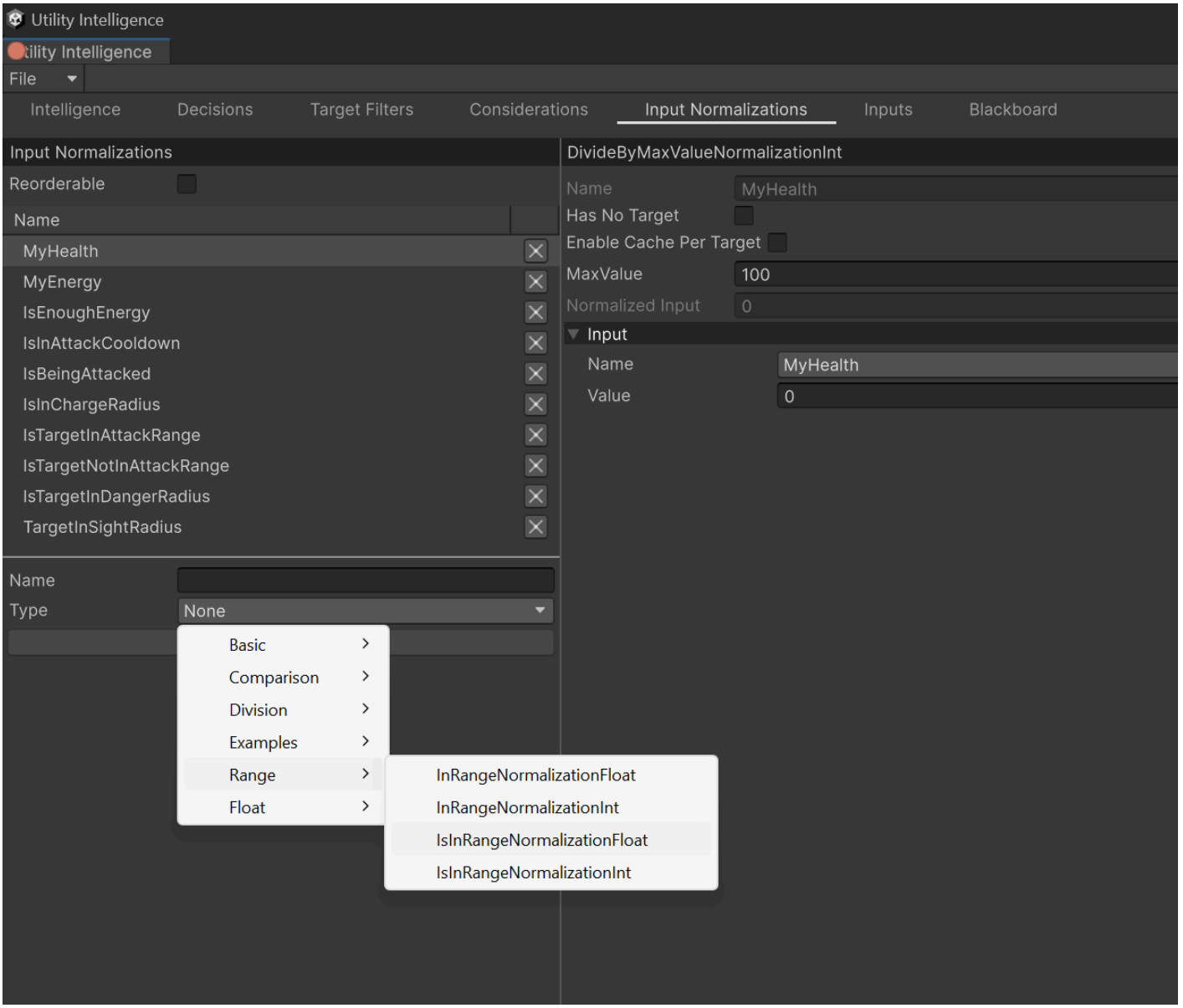


Consideration Tab

In **Consideration Tab**, you can create new considerations and select input normalizations and response curves for your considerations. Besides that, you can adjust the **input values** and **response curves** to observe how they affect the consideration scores:

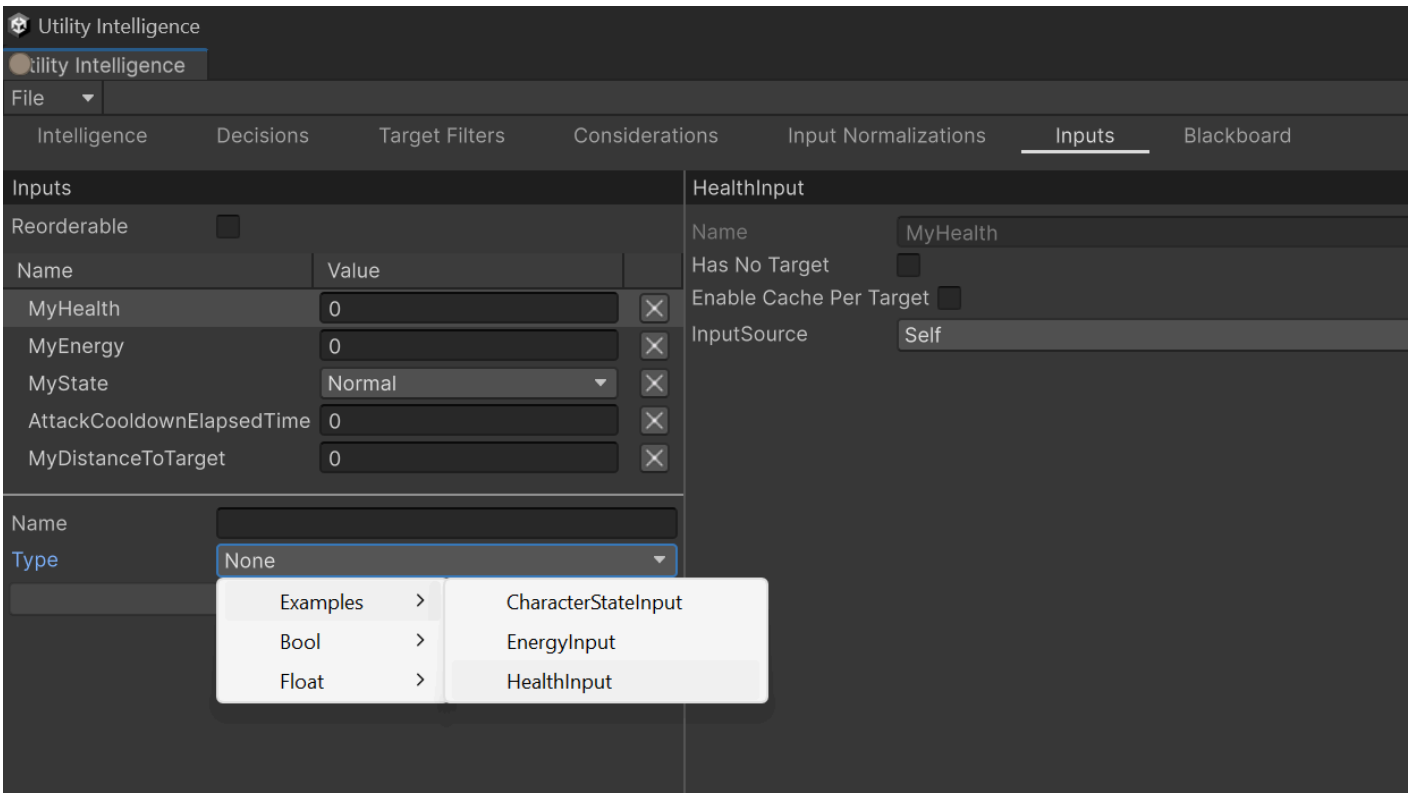
Input Normalization Tab

In **Input Normalization Tab**, you can create new input normalizations and select inputs for your input normalizations.



Input Tab

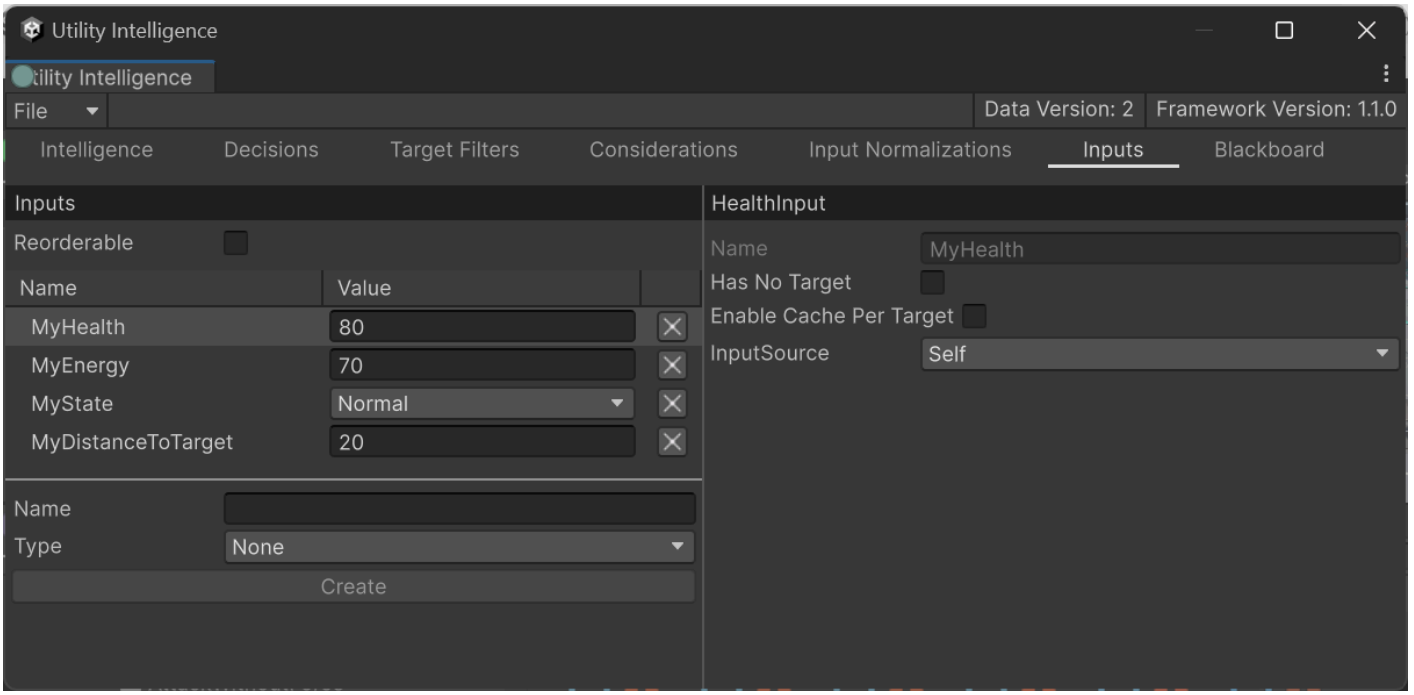
In **Input Tab**, you can add new inputs to the intelligence assets.



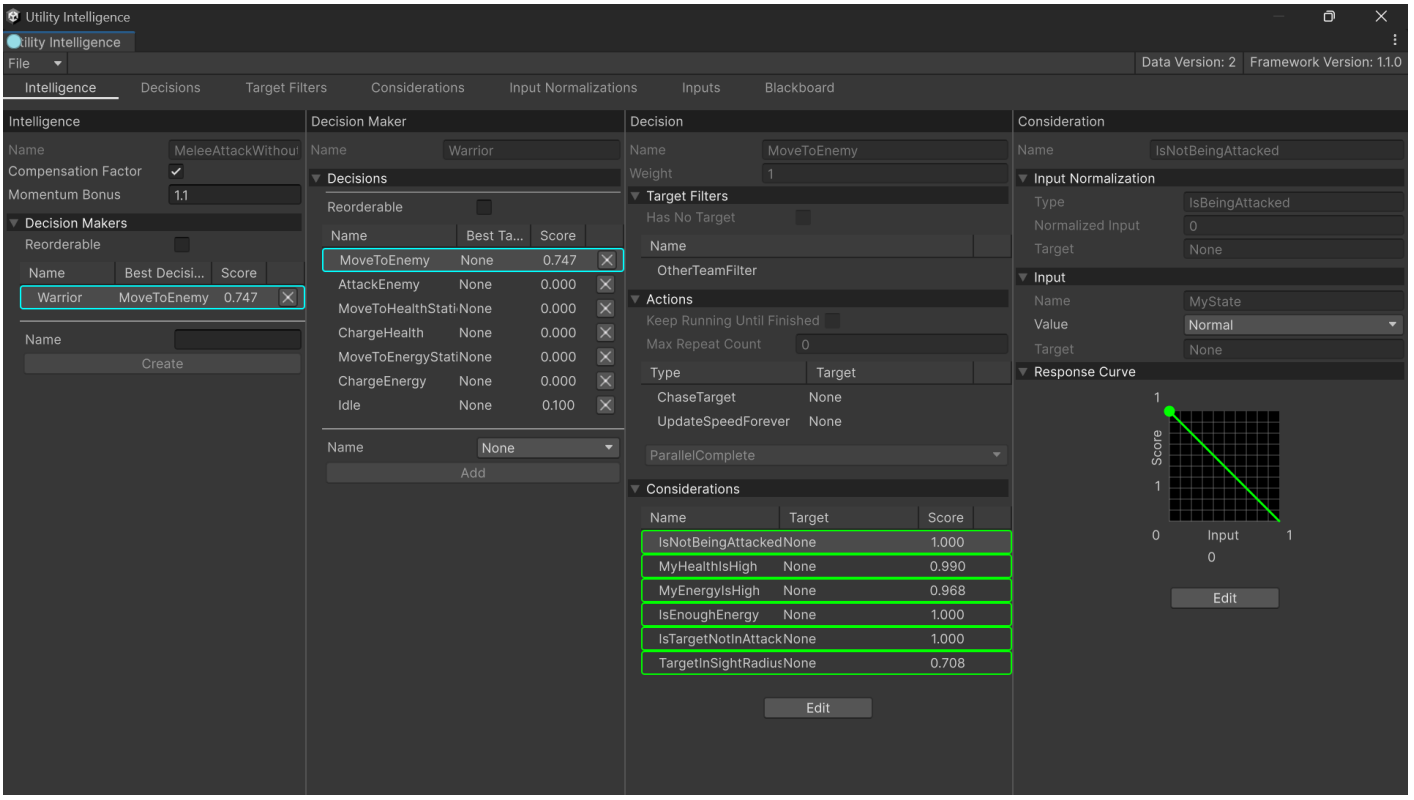
Tip

You can adjust the input values in the editor to observe how these changes affect the statuses of considerations and decisions. For further details, read Status Preview

For example, if you set the input values in the intelligence asset: **MeleeAttackWithoutForce** (in our examples) as follows:

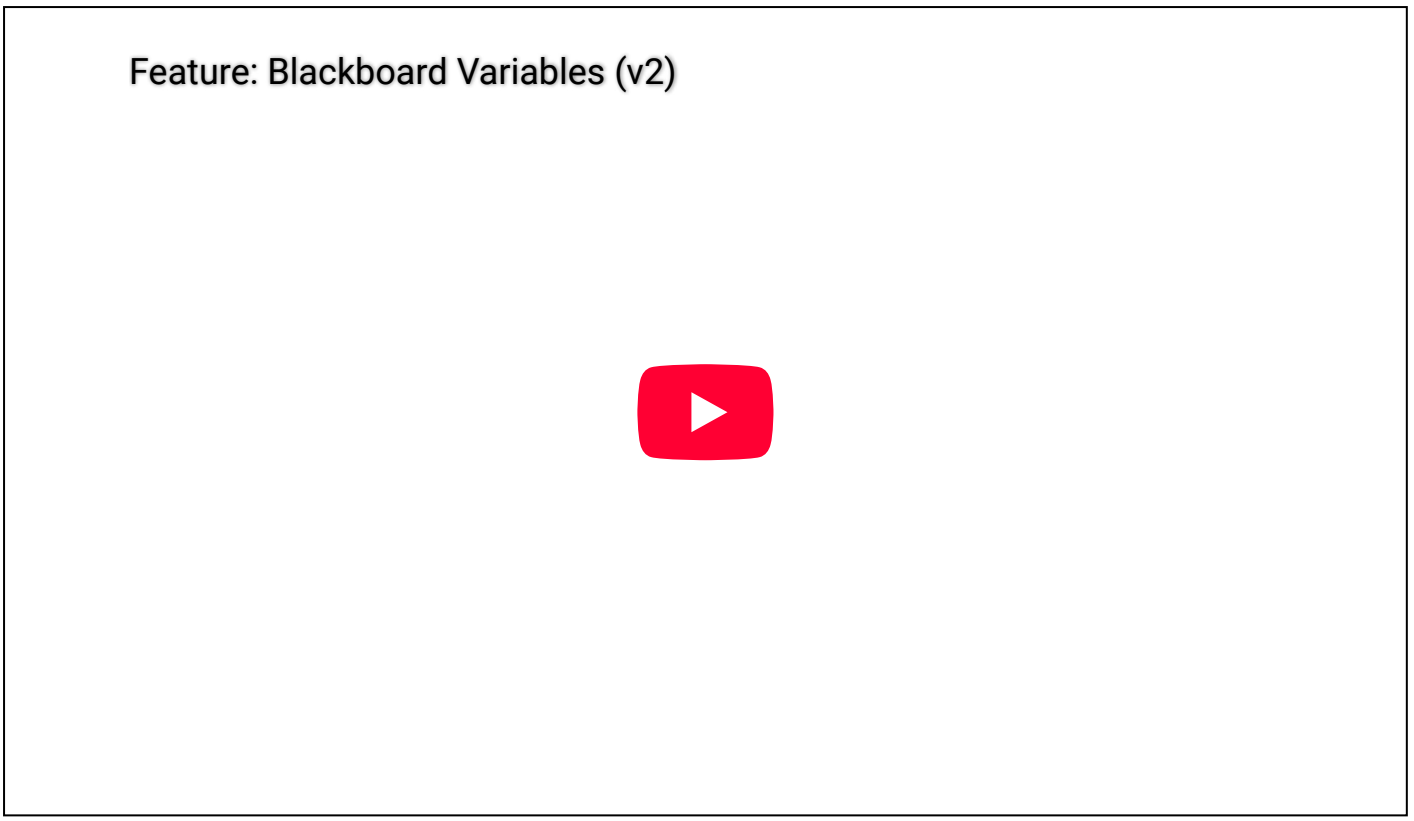


Then you will notice that the decision: **MoveToEnemy** is selected in the **Intelligence Tab**. This means you can determine which decision will be chosen based on the current input values without needing to play your game. Therefore, you will have more time to design your AIs.



Blackboard Tab

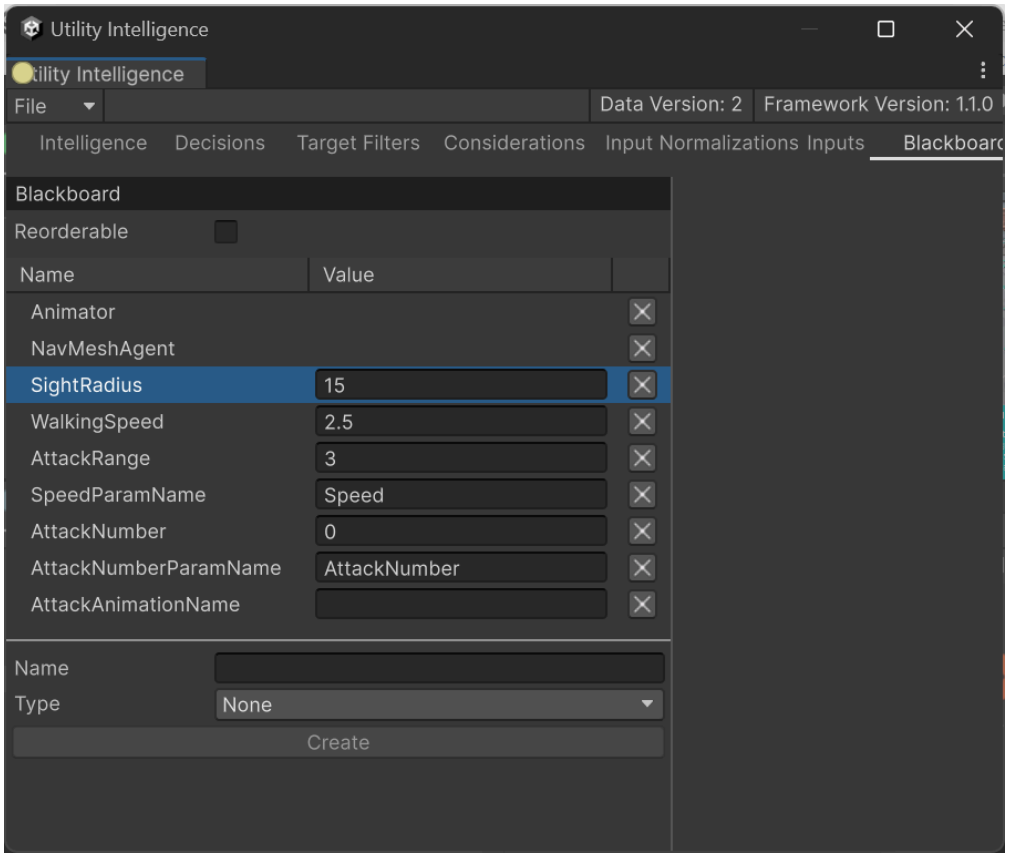
In **Blackboard Tab**, you can add variables to share information between multiple components within the agent, such as inputs, input normalizations target filters, and action tasks.



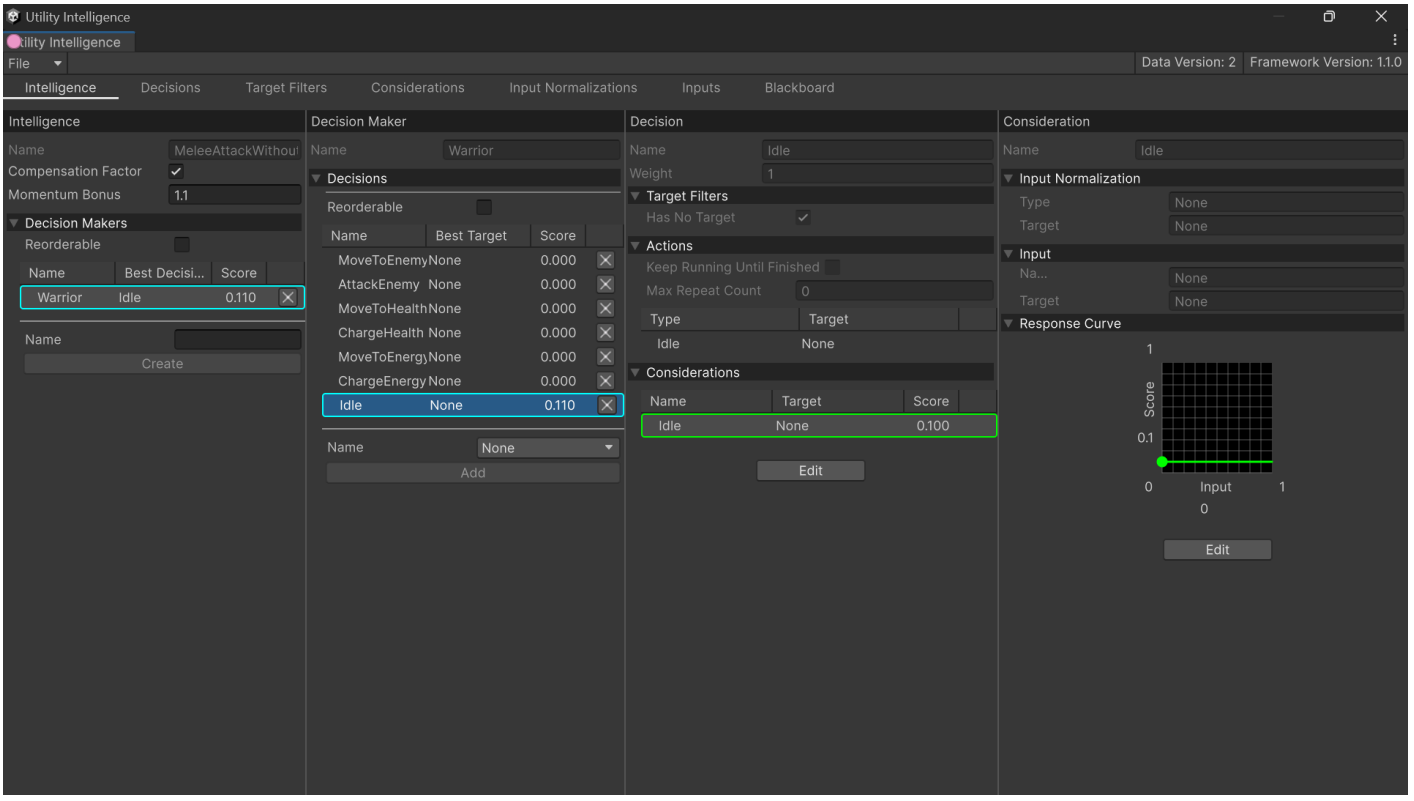
Tip

If some of your blackboard variables are used by one of the inputs or input normalizations, then changing the values of those variables will also affect the statuses of considerations and decisions, just like input values.

Continuing with the example from Input Tab, if you change the sight radius to **15** (the original value is **40**):

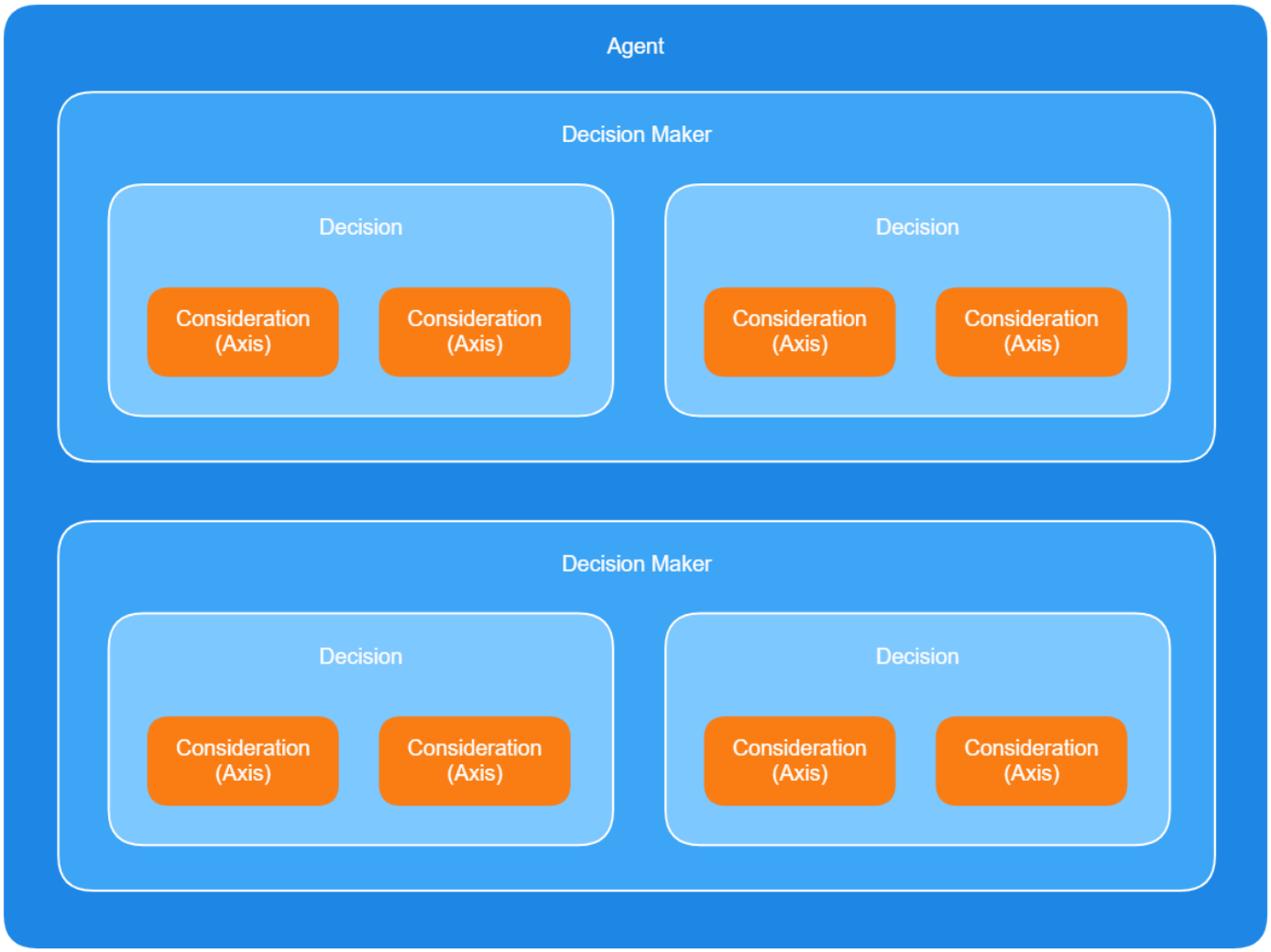


Then in the Intelligence Tab, the selected decision will change from **MoveToEnemy** to **Idle** because the enemy is out of the agent's sight:



Decision Makers

In **Utility Intelligence**, a decision maker contains a list of decisions, and the responsibility of each decision maker is to select the best decision from them based on the current situation. Additionally, each utility agent can contain multiple decision makers.



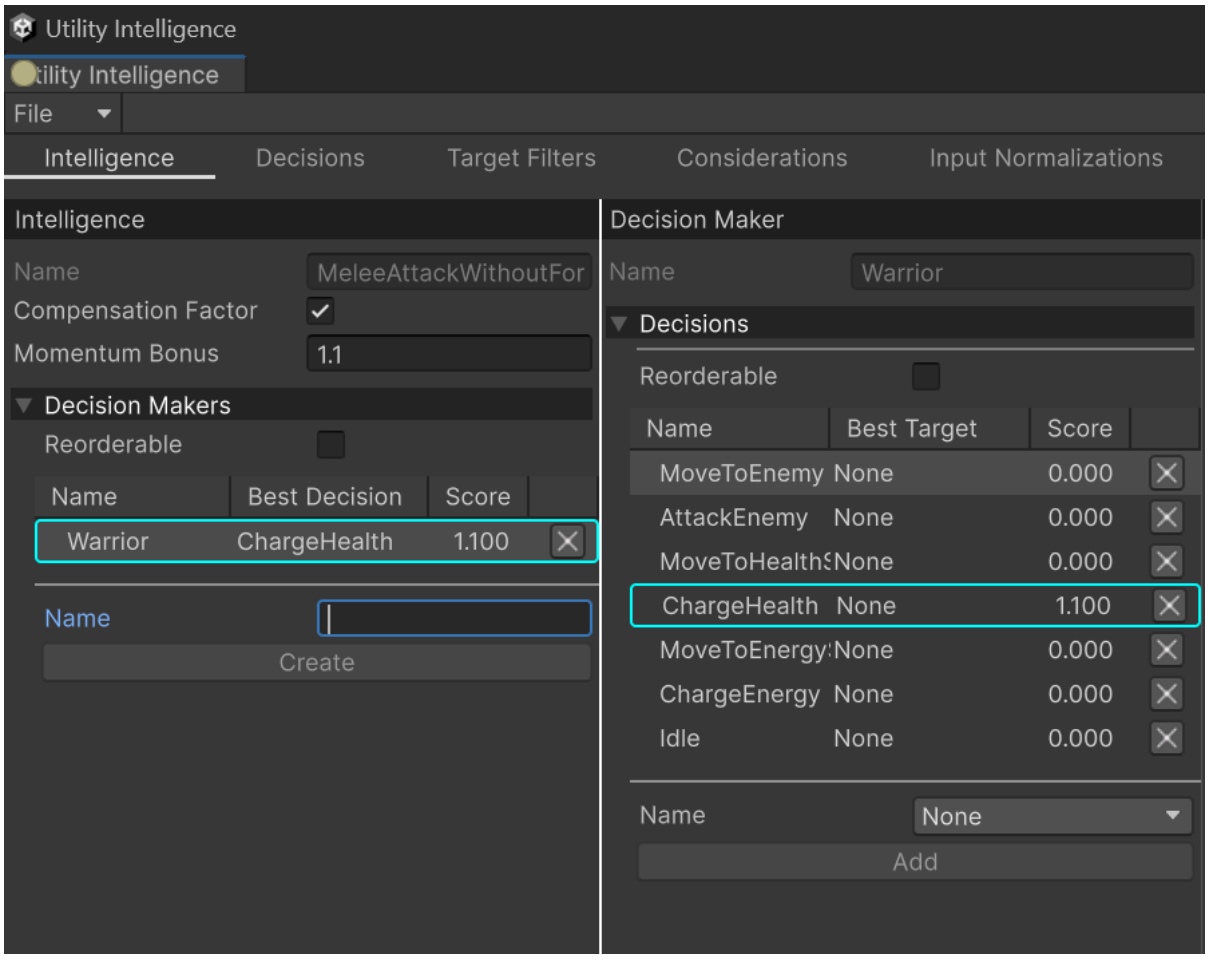
Understanding how the decision-making process works

Here's how the **decision-making process** of a utility agent works:

1. For each decision maker, the utility agent calculates the scores of all attached decisions and selects the best decision.
2. Afterwards, the utiltiy agent compares the scores of the best decisions from each decision maker with each other, and the winner is the decision with the highest score.

Creating Decision Makers

To create a decision maker, you need to go to the Intelligence Tab, fill in the **Name** Field, and then click the **Create** button:



After creating a decision maker, you can add Decisions to it and monitor which decision will be chosen as the best one based on the current situation.

Decision Maker Statuses

At runtime, decision makers have 4 statuses:

: Running

: Success

: Failed

: Inactive

At editor time, decision makers have 2 statuses:

: Selected

: Unselected

Decisions

In **Utility Intelligence**, each decision has:

- A list of Target Filters: They are used to filter targets for the decision.
- A list of Considerations: They are used to calculate the score of the decision.
- A list of Action Tasks: They will be executed by the egent if the decision is chosen.

Understanding how decisions work

Since a decision is scored per target, and any Utility Entity (all GameObjects with `UtilityEntityController` or `UtilityAgentController` attached) in the Utility World could be a target of the decision, we need a way to filter targets to ensure that only appropriate targets are considered. This is the job of Target Filters.

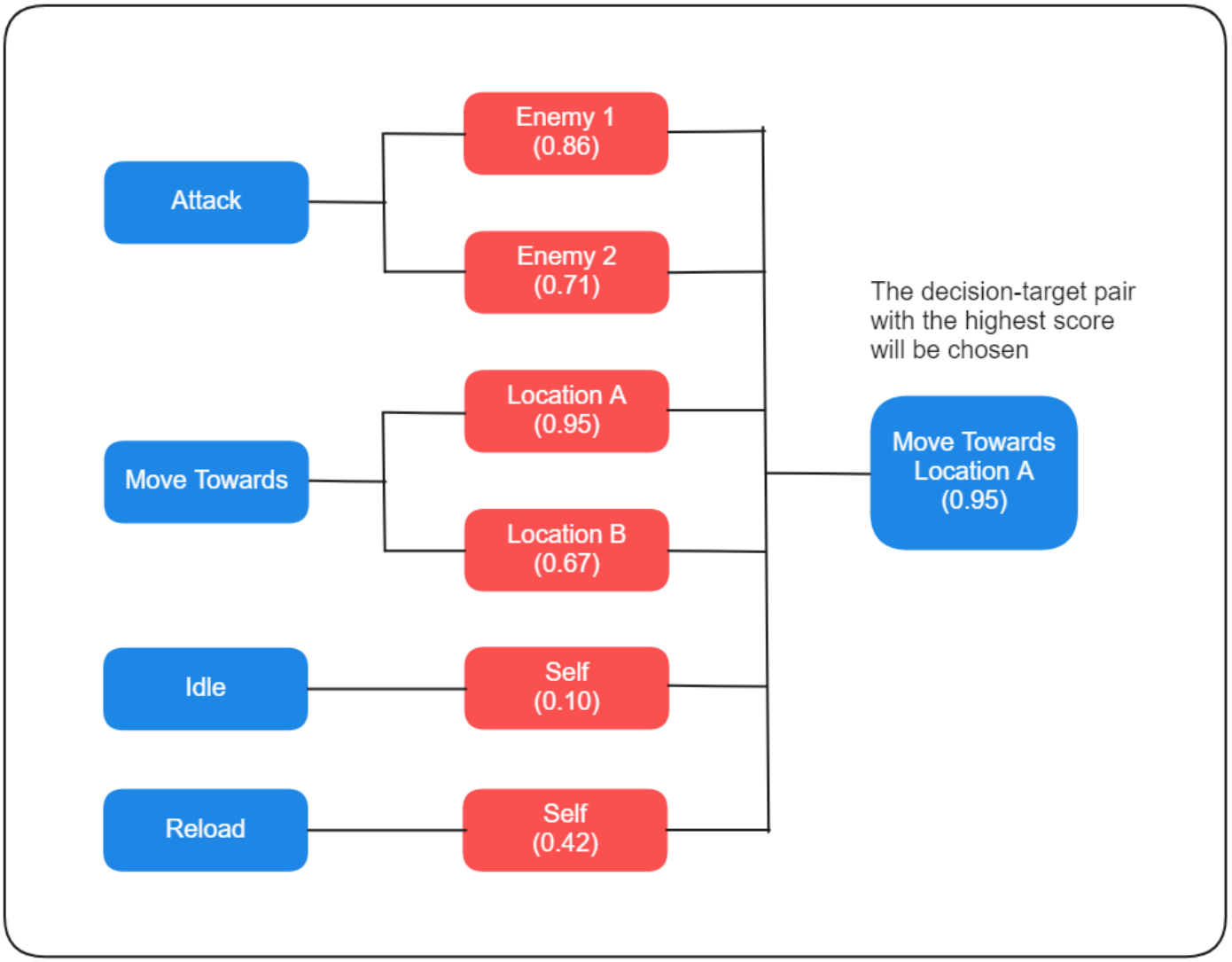
After finding appropriate targets, all Considerations of the decision will be evaluated for each target to calculate the score of each decision-target pair. Then the score of each pair is multiplied with the Decision weight to get the final score.

Finally, the best decision-target pair with the highest score will be chosen and the agent will execute all Action Tasks attached to the decision, either in **Sequence** or in **Parallel**.

Decisions are scored per target

A decision may or may not have targets. However:

1. If it has targets, it will be **scored per target**. Afterward, **Utility Intelligence** will compare the scores of all the decision-target pairs with each other and select the pair with the highest score.
2. If it does not have targets, it will be scored only once, and that score is the final score of the decision.



Oscillation between decision-target pairs

When using Utility AI, there may be a scenario where decision-target pairs with similar scores oscillate back and forth as their scores rise and fall. This leads to the agent constantly changing its decision and target. Currently, there are four ways to address this issue:

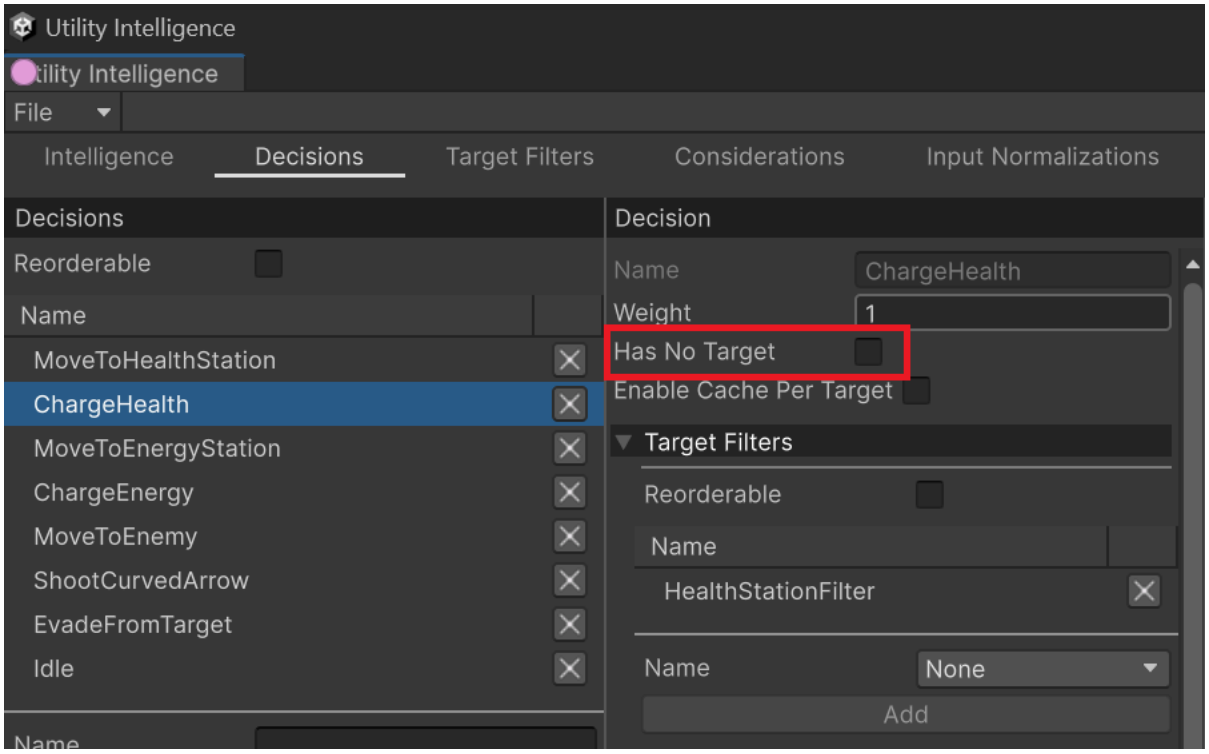
1. Enable the Momentum Bonus option to add a **bonus** to the last chosen decision-target pair in the next decision-making round.
   - This will prioritize the last decision-target pair over the others, thereby eliminating the oscillation.
2. Increase the weight of the decision that you want to prioritize. For example, let's say 2 or 3 instead of just 1.
   - This will prioritize one decision over the others, reducing the oscillation.
3. Enable Keep Running Until Finished option to prevent the agent from making a new decision while an important task is running.
   - When the agent is performing an important task, such as **AttackPlayer**, **ChargeHealth**, **ReloadAmmunition**, and you don't want it to be interrupted, you can enable this option to prevent the

agent from switching to another decision while that task is running.

4. Add more considerations to each decision.
    - This will introduce more variation to the score-calculation process, increasing the chances that the competing decision will consistently win (or lose) and thereby reducing the oscillation.

Has No Target

A decision may or may not have targets. You can specify whether it has targets or not by checking/unchecking the **HasNoTarget** toggle in the **Decision Tab**:



- If the **HasNoTarget** toggle is checked:
    - The target filter list will be hidden because it is no longer necessary.
    - The decision will be considered as having no target, and will be scored only once without targets.
- If the **HasNoTarget** toggle is unchecked:
    - If the target filter list is empty:
        - All utility entities in the same utility world will be considered as targets for the decision, and the decision will be scored per target.
    - If the target filter list is not empty:
        - If the filtered targets **> 0**, the decision will be scored per target.
        - If the filtered targets **= 0**, the decision score will be **0**

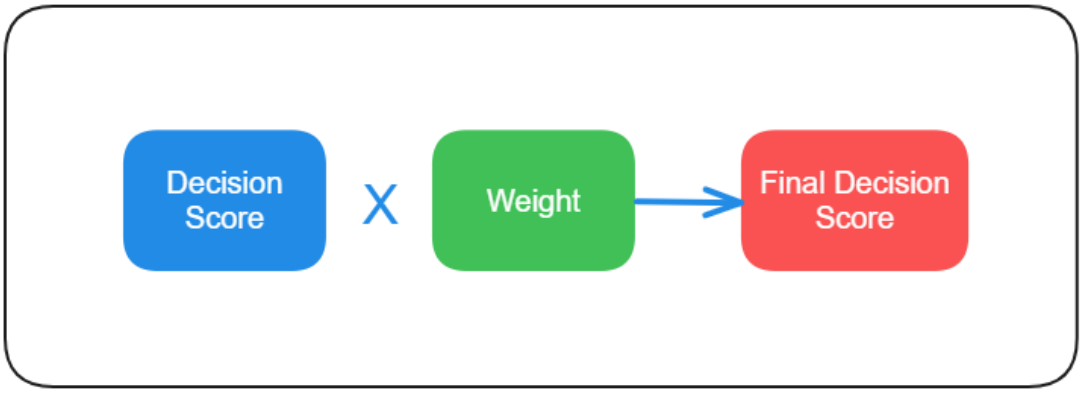
Decision Weight

In **Utility Intelligence**, you can control the prioritization of each decision by adjusting its **Decision Weight**.

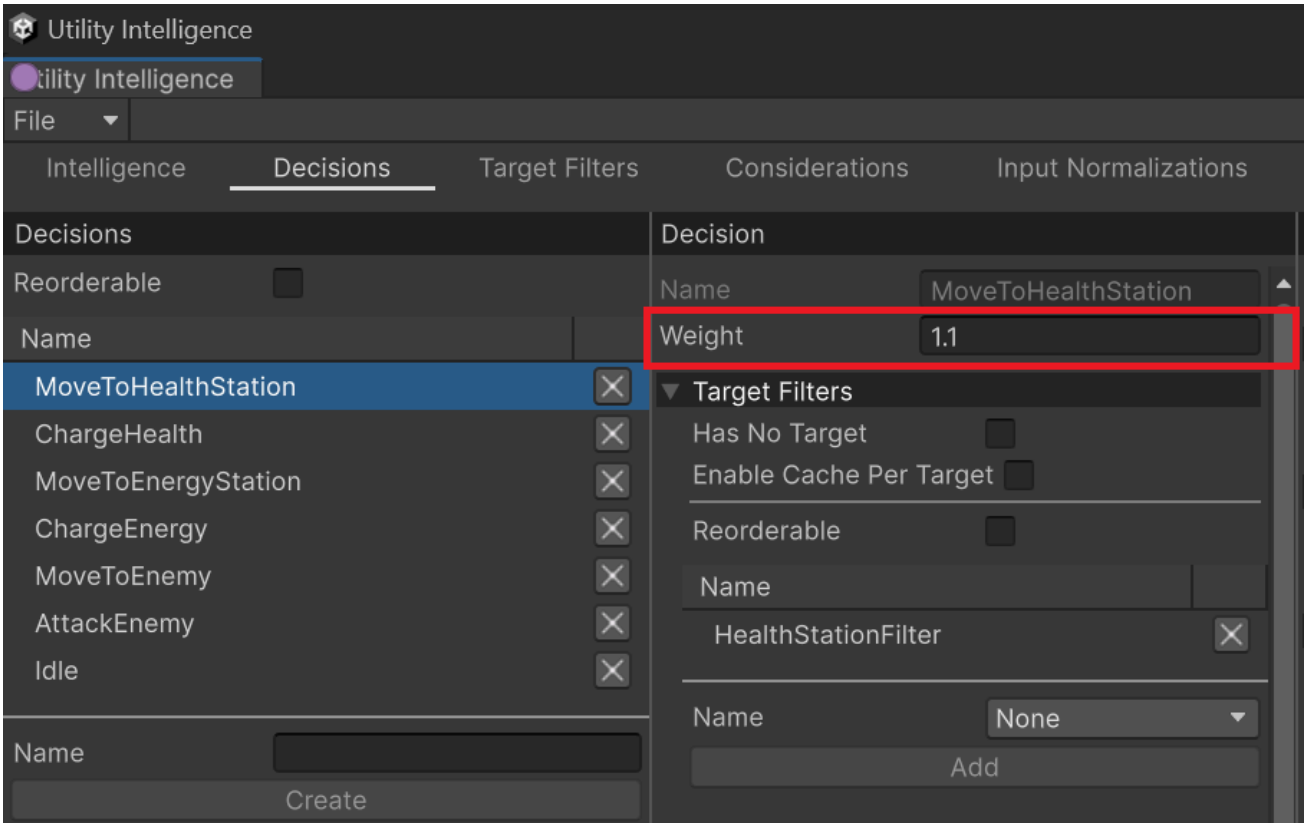
For example, you can organize your decisions into multiple layers like the following:

- Normal Layer's Weight: 1.0
- Combat Layer's Weight: 2.0
- Urgent Layer's Weight: 3.0

The decision weight will then be multiplied by the decision score to get the final decision score:



You can change the weight of a decision in the **Decision Tab**:



Info

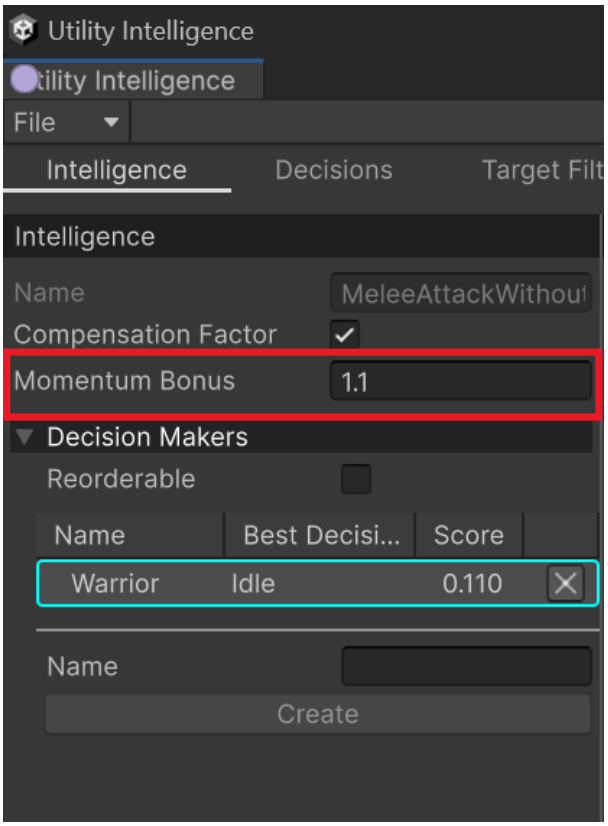
- The default value of **Decision Weight** is **1.0**.
- You can adjust the weight of a decision to a value lower than **1.0** to decrease the priority of that decision.

Tip

- You can adjust the weight of decisions to reduce oscillation between nearly equal decision-target pairs.

Momentum Bonus

In **Utility Intelligence**, you can prioritize the **last chosen** decision-target pair in the **next** decision-making round by increasing the **Momentum Bonus**:



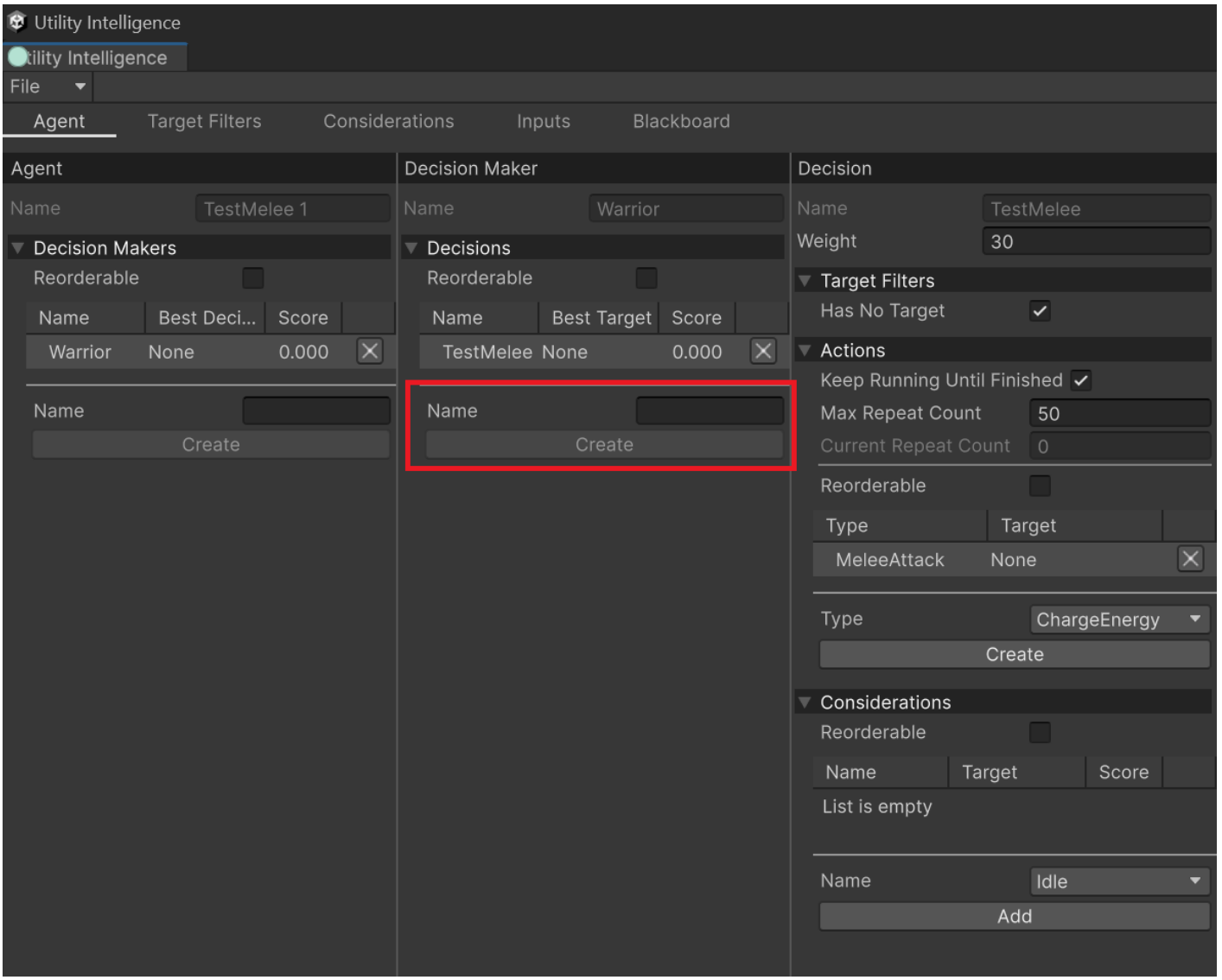
In the next decision-making round, the **last chosen** decision-target pair will be prioritized by multiplying its score by the **Momentum Bonus**, increasing its chances of winning and thereby reducing oscillation between nearly equal decision-target pairs.

Info

- The default value of **Momentum Bonus** is **1.1**.
- You can adjust the **Momentum Bonus** to a value lower than **1.0** to decrease the priority of the **last chosen** decision-target pair.

Creating Decisions

To create a new decision, you need to go to the Intelligence Tab, fill in the **Name** field, and then click the **Create** button:



After creating a decision, you can add Considerations to the decision and observe how they affect the decision score. Additionally, you can add target filters and action tasks to the decision to determine which actions will be executed with its targets if the decision is selected at runtime.

Decision Statuses

At runtime, decisions have 4 statuses:

 : Running

 : Success

 : Failed

: Inactive

At editor time, decision only have 2 statuses:

 : Selected

: Unselected

Target Filters

Target Filters are used to filter targets for the current decision.

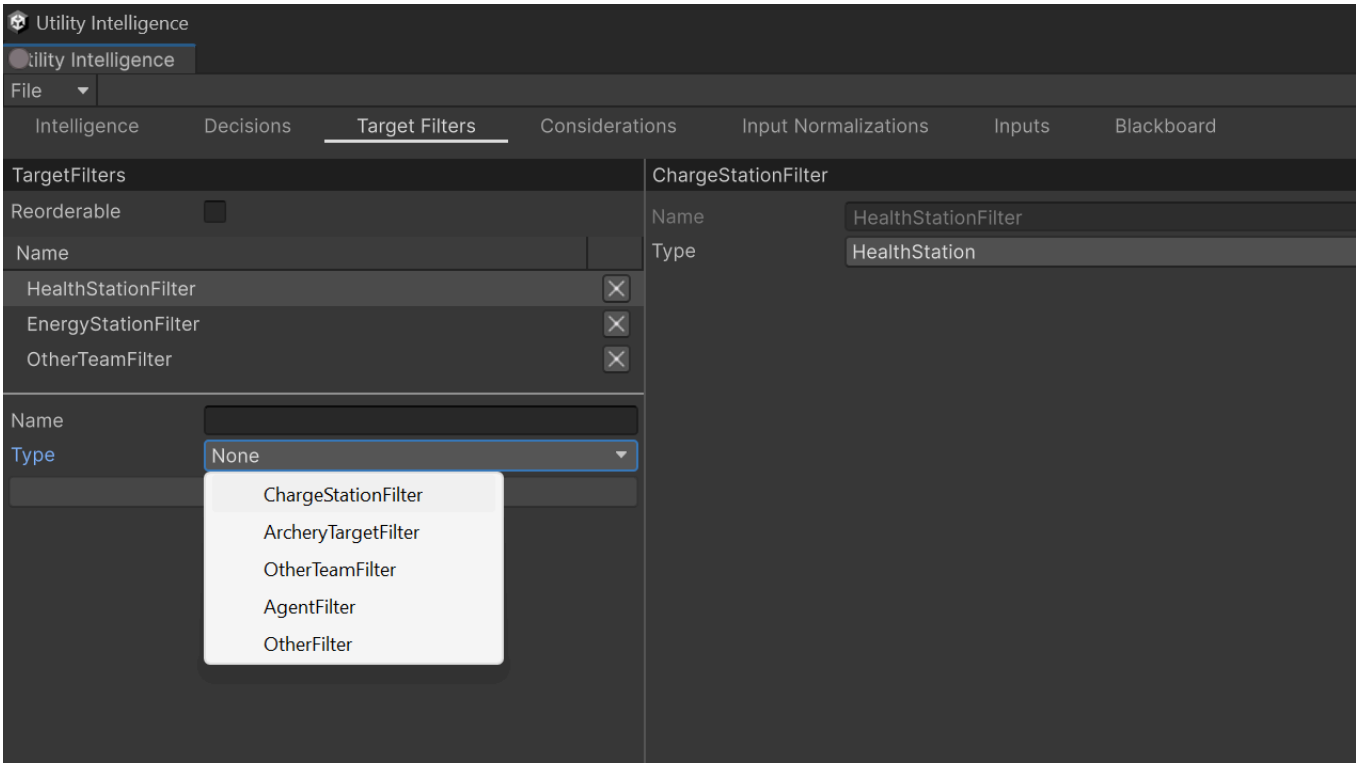
> **Note**
>
> - A decision may or may not have targets, so target filters are optional.
> - You can enable/disable target filters of a decision by checking/unchecking the HasNoTarget toggle in the **Decision Tab**.

Creating Target Filters

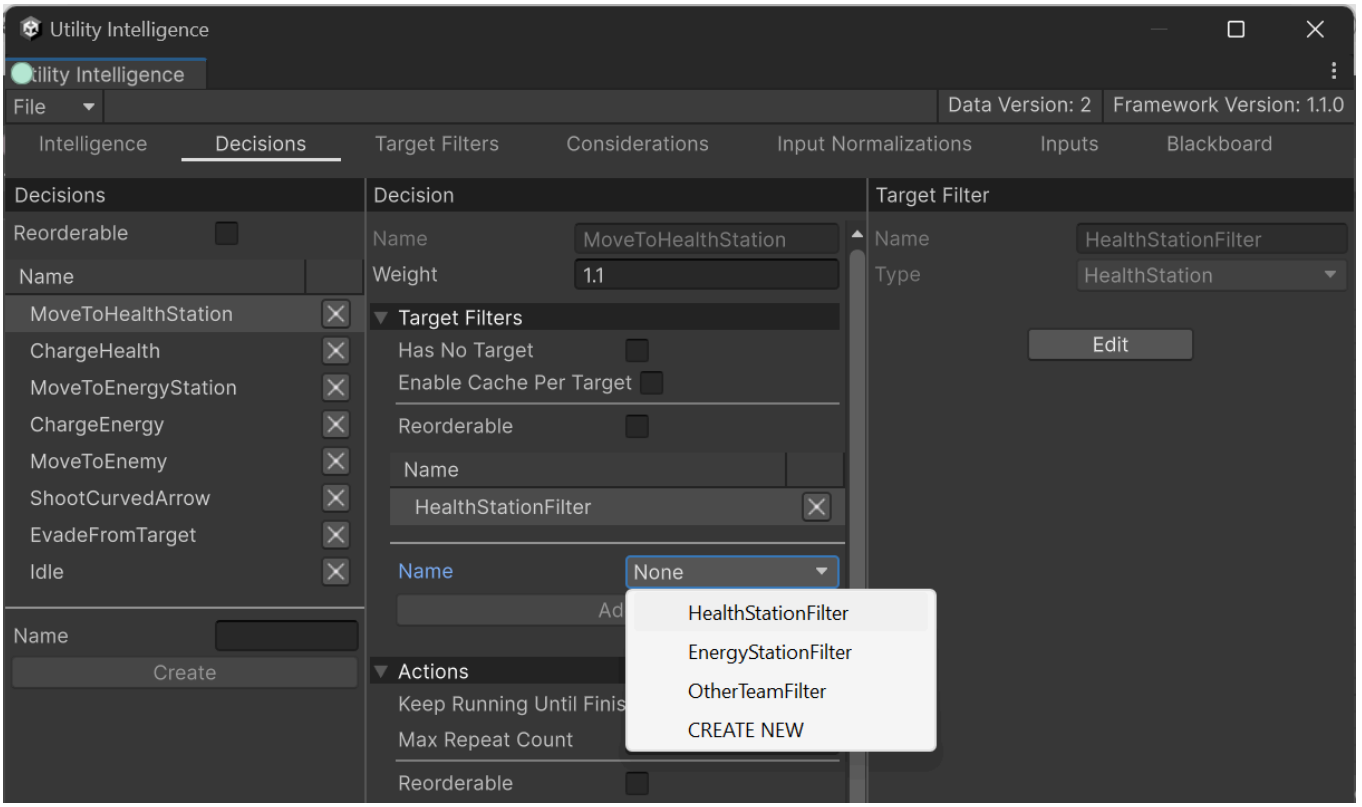
1. To create a new target filter, define a new class that inherits from `TargetFilter` and override the `OnFilterTarget` method:

```
public class ChargeStationFilter : TargetFilter
{
    public ChargeStationType Type;

    protected override bool OnFilterTarget(UtilityEntity target)
    {
        return target.EntityFacade is ChargeStation station && station.Type == Type;
    }
}
```

2. To add the the target filter to the intelligence asset, go to the **Target Filter Tab**, select the target filter type, give it a name, and then click the **Create** button:



3. To attach the target filter to a decision, select the decision in the **Decision Tab**, choose the target filter's name, and then click the **Add** button:



Adding Parameter Fields

There are many cases when you need to add parameters to an target filer to customize how it filter targets. To achieve this, you need to declare these parameters as public fields in your target filters. Here an example of how to do this:

```
public class TeamFilter : TargetFilter
{
    public Team Team;
    protected override bool OnFilterTarget(UtilityEntity target)
    {
        if (target.EntityFacade is Character targetCharacter)
        {
            return targetCharacter.Team == this.Team;
        }

        return false;
    }
}
```

Supported Field Types

Currently, only the supported field types can be serialized to **JSON** and adjusted using the Utility Intelligence Editor. Therefore, you should use these types when declaring parameter fields for your target filters.

Built-in Target Filters

Currently, we provides these built-in target filters:

- **OtherFilter**: Filters out the current agent, leaving other entities as targets.
- **AgentFilter**: Filters out entities that are not utility agents, leaving only utility agents as targets.

Action Tasks

Action Tasks are tasks that the agent has to execute if the attached decision has been selected. They are executed either in sequence or in parallel, depending on the execution mode of the action list.

What is the action task system based on?

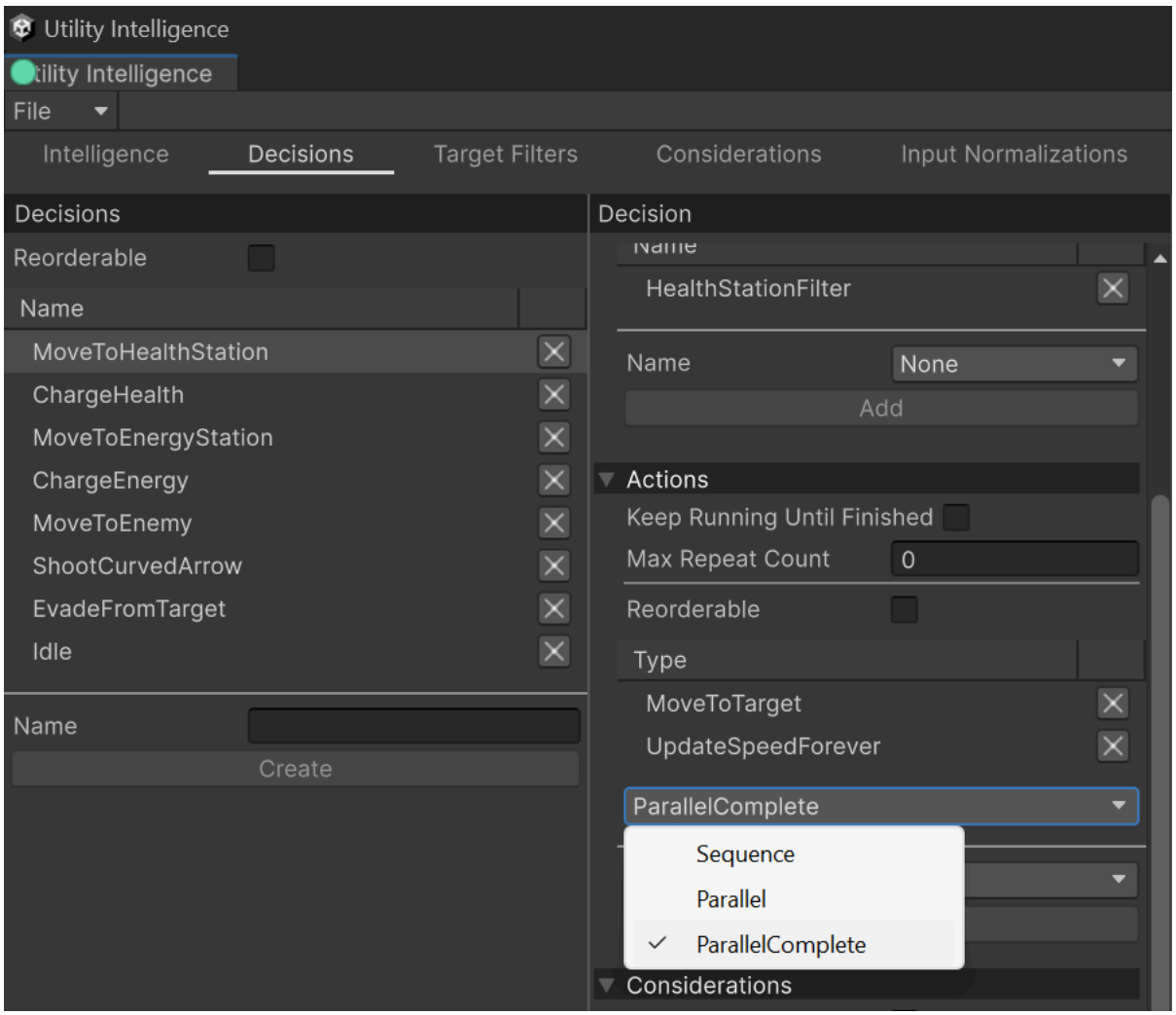
Utility Intelligence uses **Behavior Trees** to create and execute action tasks. Basically, the action task system is a simplified **Behavior Tree**. It includes some popular nodes such as **Repeater**, **Sequencer**, and **Parallel**.

Execution Modes

After the agent finds out the best decision, it will execute the action list either in **sequence** or in **parallel**, depending on your choice. Currently, there are two execution modes for the action list:

- **Sequence**
  - The actions will be run sequentially.
  - If an action finishes in success, the agent will run the next action, and the action list will finish in success if the last action finishes in success.
  - If an action finishes in failure, the action list will finish in failure.
- **Parallel**
  - The actions will be run simultaneously.
  - The action list will finish in success if all actions are finished in success.
  - If any action finishes in failure, other actions will be aborted and the action list will finish in failure.
- **ParallelComplete**
  - The actions will be run simultaneously.
  - If any action finishes in success or failure, other actions will be aborted and the action list will return the child status immediately.

You can choose the execution mode you want by selecting it from the action execution dropdown menu in the **Decision Tab**.

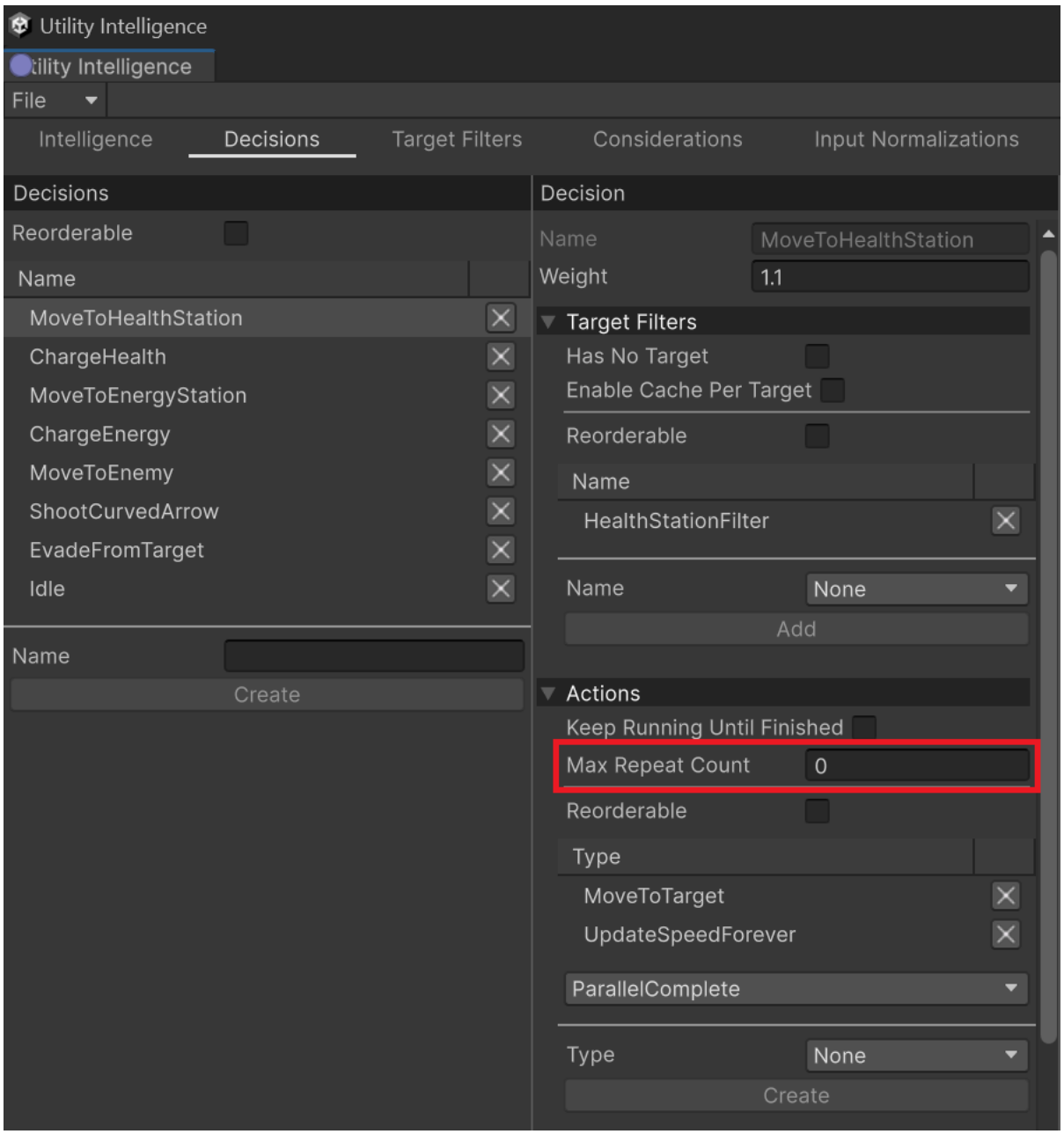


Max Repeat Count

It is the number of times to repeat the action list.

> **Note**
>
> - The action list will only repeat if it is finished in success.
> - If `MaxRepeatCount` <= 0 it will repeat forever until it returns failure.

You can change `MaxRepeatCount` of the action list here:



Keep Running Until Finished

In case you want to prevent the current agent from making a new decision while the action list is running, you can check the option: **Keep Running Until Finished** in the **Action List Editor**.

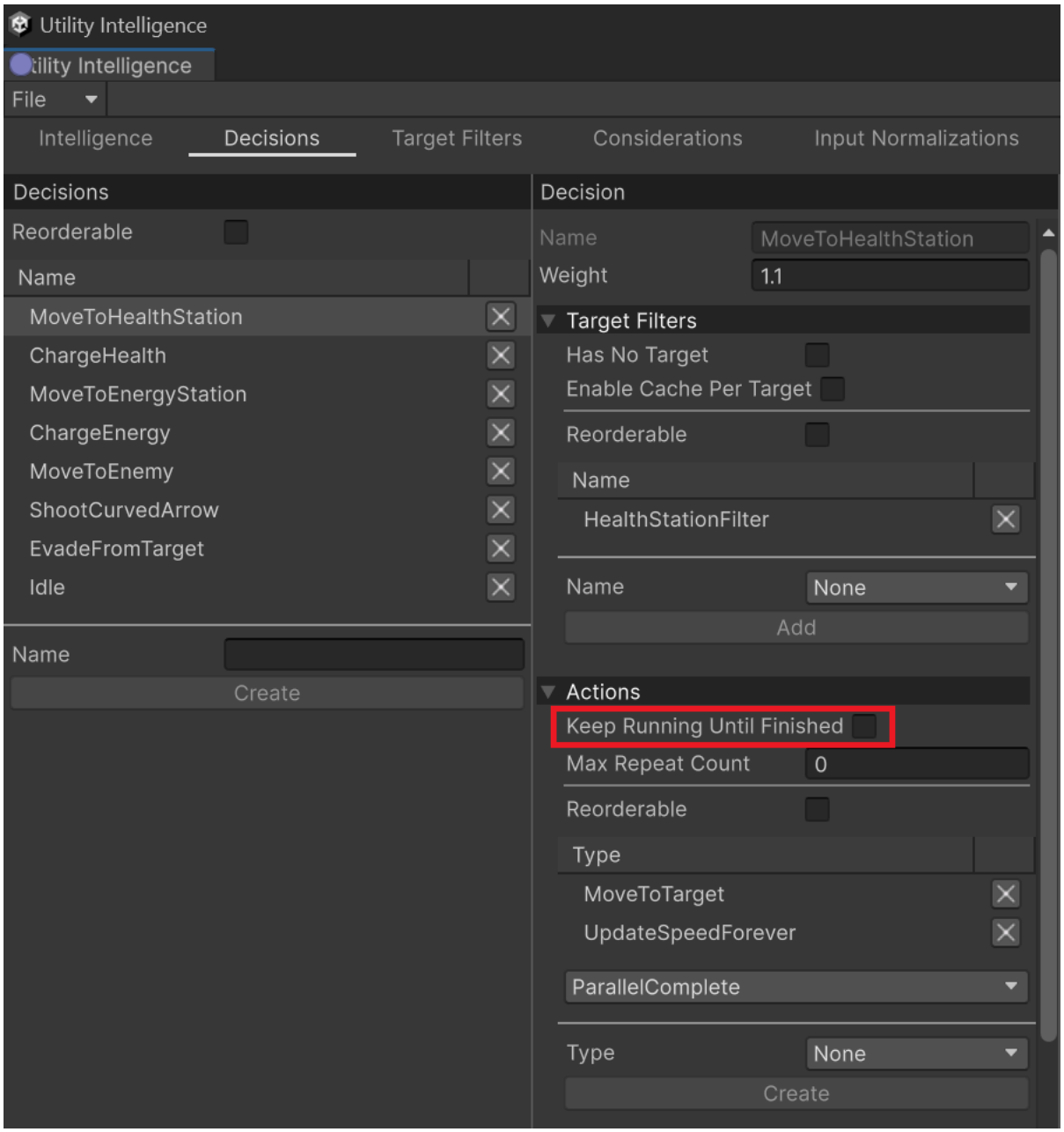
Tip

- By enabling this option for important decisions, such as **AttackEnemy**, **ChargeHealth**, and **ReloadAmmunition**, it stops the agent from getting distracted by other non-important decisions. This helps reduce the oscillation between these important decisions and other non-important ones.
- For example, with **AttackEnemy** decision, you should enable this option because the agent needs to finish the attack before switching to another decision, such as **RunAwayFromEnemy**.

Note

- If you enable this option, the agent can only change its decision after the action list is finished, regardless of whether the scores of other decisions are higher than the current one.
- For example, with **AttackEnemy** decision, the agent can only switch to another decision after each attack is finished, even if the scores of other decisions such as **RunAwayFromEnemy** or **ReloadAmmunition** are higher than **AttackEnemy**.
- Additionally, if the score of the **AttackEnemy** decision remains the highest after each attack, the agent will keep running this decision.

To enable/disable **Keep Running Until Finished** option, you need to check/uncheck it in the **Action List Editor**:

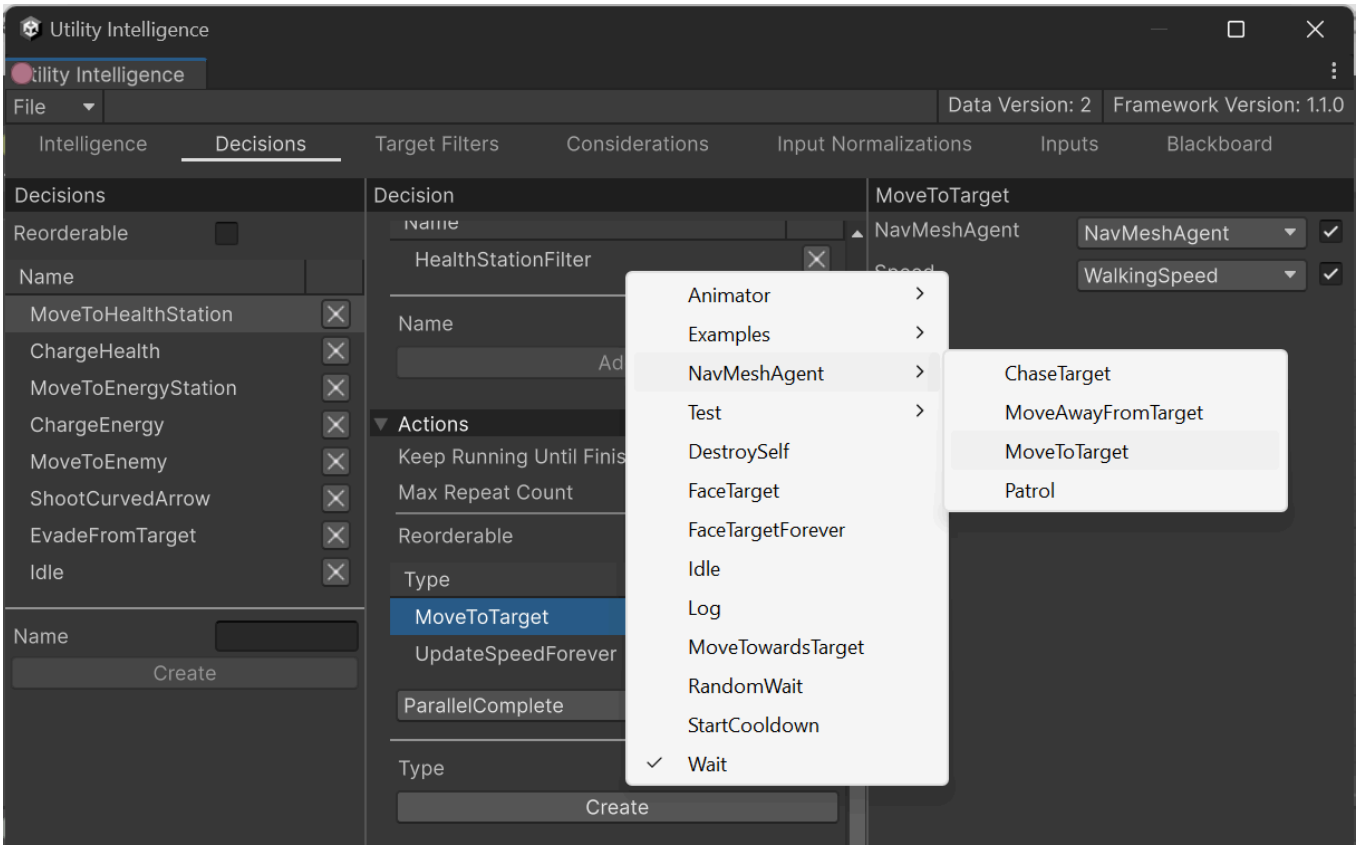


Creating Action Tasks

1. To create a new action task, define a new class that inherits from `ActionTask`:

```
public class Wait : ActionTask
{
    private float elapsedTime;
    public VariableReference<float> WaitTime = 1.0f;

    protected override void OnStart()
    {
        elapsedTime = 0;
    }

    protected override UpdateStatus OnUpdate(float deltaTime)
    {
        elapsedTime += deltaTime;

        if (elapsedTime > WaitTime) return UpdateStatus.Success;
        return UpdateStatus.Running;
    }
}
```

2. To assign the action task to a decision, select the decision in the **Decision Tab**, choose the action type, and then click the **Create** button:



Adding Parameter Fields

There are many cases when you need to add parameters to an action task to customize how it works. To achieve this, you need to declare these parameters as public fields in your action tasks. Here are some examples of how to do this:

```
[Category("Examples")]
public class StartMeleeAttack : ActionTask
{
    public MeleeAttackType AttackType;
    public int AttackDamage;
    public int AttackForce;
    public int ConsumeEnergy;

    public VariableReference<float> AttackRange;
    public VariableReference<int> AttackNumber;
    public VariableReference<string> AttackAnimationName;
}
```

```
[Category("Examples")]
public class StartRangedAttack : ActionTask
{
    public RangedAttackType AttackType;
    public int ConsumeEnergy;
    public int AttackDamage;
    public int ProjectileSpeed;
    public float MaxCurvedHeight;
}
```

Supported Field Types

Currently, only the supported field types can be serialized to **JSON** and adjusted using the Utility Intelligence Editor. Therefore, you should use these types when declaring parameter fields for your action tasks.

Action Task Statuses

At runtime, action tasks have 4 statuses:

: Running

: Success

: Failed

: Aborted

Built-in Action Tasks

Currently, **Utility Intelligence** provides these built-in action tasks:

- **Idle**: Does nothing.
  - Always returns `UpdateStatus.Running`.
- **Log**: Logs a message to the console.
- **Wait**: Waits until a wait time has passed.
  - The wait time is specified in the `WaitTime` variable.
  - Returns `UpdateStatus.Success` when the wait time has passed, otherwise, returns `UpdateStatus.Running`.
- **RandomWait**: Waits until a wait time has passed.
  - The wait time is chosen randomly between the `WaitTimeMin` and `WaitTimeMax` variables.
  - Returns `UpdateStatus.Success` when the wait time has passed, otherwise, returns `UpdateStatus.Running`.
- **DestroySelf**: Safely destroys the current agent.
- **MoveTowardsTarget**: Moves to wards the target.
  - Uses `Vector3.MoveTowards` to move the agent towards the target.
  - Returns `UpdateStatus.Success` when the agent has reached the target, otherwise, returns `UpdateStatus.Running`.
- **StartCooldown**: Starts a cooldown.
  - The start time of the cooldown is stored in the `CooldownStartTime` variable, which is used by `CooldownElapsedTimeInput` and `IsInCooldownNormalization` to determine if the agent is within the cooldown duration.
- Animator
  - **SetBool**: Set the value of the boolean parameter specified by `ParamName`.
  - **SetFloat**: Set the value of the float parameter specified by `ParamName`.
  - **SetInteger**: Set the value of the integer parameter specified by `ParamName`.
  - **SetTrigger**: Set the value of the trigger parameter specified by `ParamName`.
  - **WaitUntilAnimationFinished**: Waits until the specified animation is finished.
    - Returns `UpdateStatus.Success` if the animation specified by `AnimationName` has the normalized time greater than `FinishedNormalizedTime`, otherwise, returns `UpdateStatus.Running`.
- NavMeshAgent
  - **ChaseTarget**: Chases the target.
    - The target position is updated every frame.
    - Returns `UpdateStatus.Success` when the agent has reached the target, otherwise, returns `UpdateStatus.Running`.
  - **MoveToTarget**: Moves to the target.
    - The target position is updated only once at the start.

    - Returns `UpdateStatus.Success` when the agent has reached the target, otherwise, returns `UpdateStatus.Running`.
- **MoveAwayFromTarget**: Moves away from the target.
    - It will choose a destination at a distance specified in the `DistanceToNextPoint` variable from the current agent with the direction based on an enum called `DirectionPriority`.
    - Returns `UpdateStatus.Success` when the agent has reached the destination, otherwise, returns `UpdateStatus.Running`.
- **Patrol**: Patrols around the waypoints.
    - It will move to the next way point in the `Waypoints` variable if it has reached the current one.
    - Always returns `UpdateStatus.Running`.

- **FaceTarget**: Faces the target.
    - Returns `UpdateStatus.Success` right after the first update.
- **FaceTargetForever**. Faces the target forever.
    - Always returns `UpdateStatus.Running`.

Properties and Functions

Properties

Here are some useful properties that you can use in your custom tasks:

```
Transform Transform { get; private set; }

GameObject GameObject { get; private set; }

UtilityAgentController AgentController { get; private set; }
```

Functions

GetComponent Functions

You can get any component attached to the GameObject by calling these functions:

```
T GetComponent<T>()
T GetComponentInChildren<T>()
```

Coroutine functions

We provides these functions to help you start/stop coroutines from action tasks:

```
void StartCoroutine(string methodName);

Coroutine StartCoroutine(IEnumerator routine);

Coroutine StartCoroutine(string methodName, object value);

void StopCoroutine(string methodName);

void StopCoroutine(IEnumerator routine);

void StopAllCoroutines();
```

Overridable Functions

Here is the list of functions you could override to make your actions works as you want:

- **Lifecycle Functions**:

```
void OnAwake();

void OnStart();

Status OnUpdate();

void OnLateUpdate();

void OnFixedUpdate();

//OnAbort is called when the action's target changes or when the agent makes a new decision
void OnAbort();

//OnEnd is called after the action returns a success or failure
void OnEnd();
```

- **Collision/Trigger 3D**:

```
void OnCollisionEnter(Collision collision);

void OnCollisionStay(Collision collision);

void OnCollisionExit(Collision collision);

void OnTriggerEnter(Collider other);

void OnTriggerStay(Collider other);

void OnTriggerExit(Collider other);

void OnControllerColliderHit(ControllerColliderHit hit);
```

- **Collision/Trigger 2D**:

```
void OnCollisionEnter2D(Collision2D collision);

void OnCollisionStay2D(Collision2D collision);

void OnCollisionExit2D(Collision2D collision);

void OnTriggerEnter2D(Collider2D other);

void OnTriggerStay2D(Collider2D other);

void OnTriggerExit2D(Collider2D other);
```

- **Animation**:

```
void OnAnimatorMove();

void OnAnimatorIK(int layerIndex);
```

Considerations

In **Utility Intelligence,** a **consideration (also called axis)** represents an aspect of the game world that influences the utility of a decision. And its score indicates **how appealing** the decision is based on that aspect.

For instance, imagine our agent has a decision called `AttackEnemy`, which includes a consideration caled **EnemyHealthIsLow**. Suppose the enemy's health is `20`, then the utility score of this consideration would be `0.8`, indicating high appeal to the agent. However if the agent's health rises to `60`, then the utility score decreases to `0.4`, making the decision less appealing to the agent.

> **Infinite Number of Considerations (Axes)**
>
> - You can add **an infinite number of considerations (axes)** to a decision. That's why **Dave Mark** called it: **Infinite Axis Utility System**.
>   - For more information about **Infinite Axis Utility System**, you can watch his presentations here.
> - However, the more considerations you add, the closer decision score approaches 0. To address this, we introduced Compensation Factor.

Understanding how considerations work

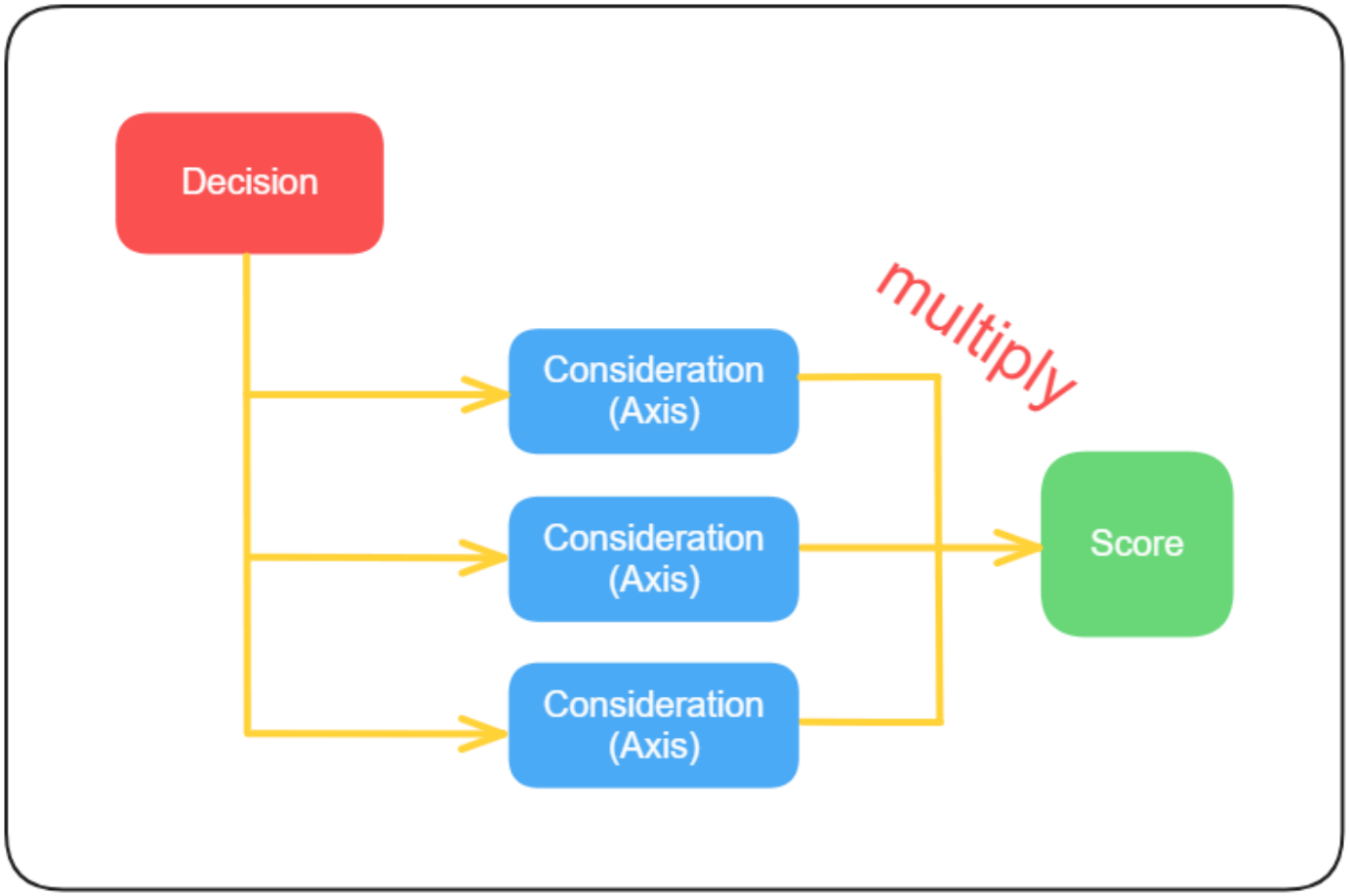
A consideration is made up of three components:

- An Inputs
- An Input Normalizations
- A Response Curve

These represent three phases to calculate the score of a consideration. In the previous example, the **EnemyHealthIsLow** consideration has the following components:

- An Inputs that returns the enemy's health.
- An Input Normalizations that normalizes the enemy's health into `[0,1]`
- A Response Curve that linearly inverts the normalized enemy's health. It returns the consideration score that indicates **how appealing** the decision is based on the **enemy's health aspect**.

Then these consideration scores will be multiplied together to get the final score of the decision. Therefore, if the score of any consideration is `0`, then the score of the decision will also be `0`.



Compensation Factor

The more considerations a decision has, the lower the score it will be due to the multiplication. For example, if a decision has 9 considerations and the score of each consideration is `0.9`, then the final score of it will be **0.9^9 = 0.387**.

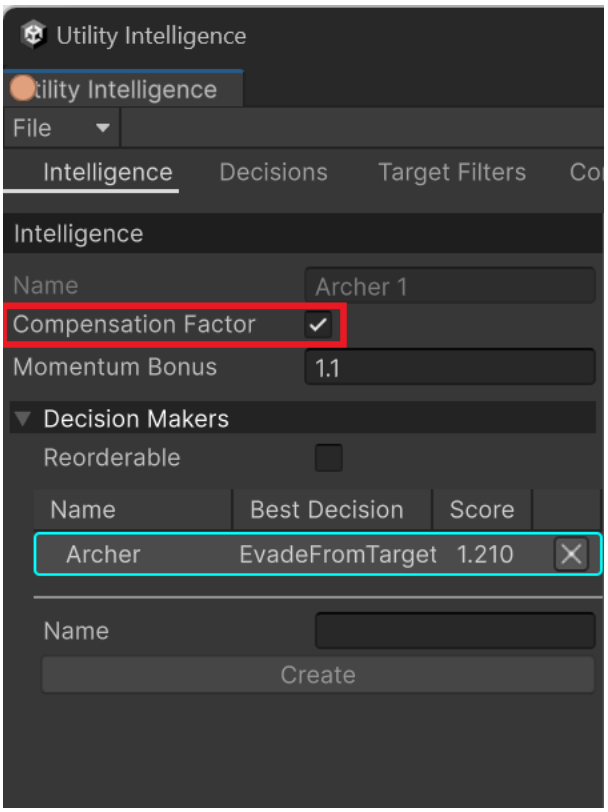
Therefore, theoretically, if a decision has an infinite number of considerations, even if the consideration scores are high, the final score of the decision will be close to `0`.

To address this issue, we added the **Compensation Factor** calculation, which takes into account the number of considerations to balance it. This calculation originally presented in Building a Better Centaur: AI at Massive Scale (9:10).

Here's how the compensation factor calculation is implemented in code:

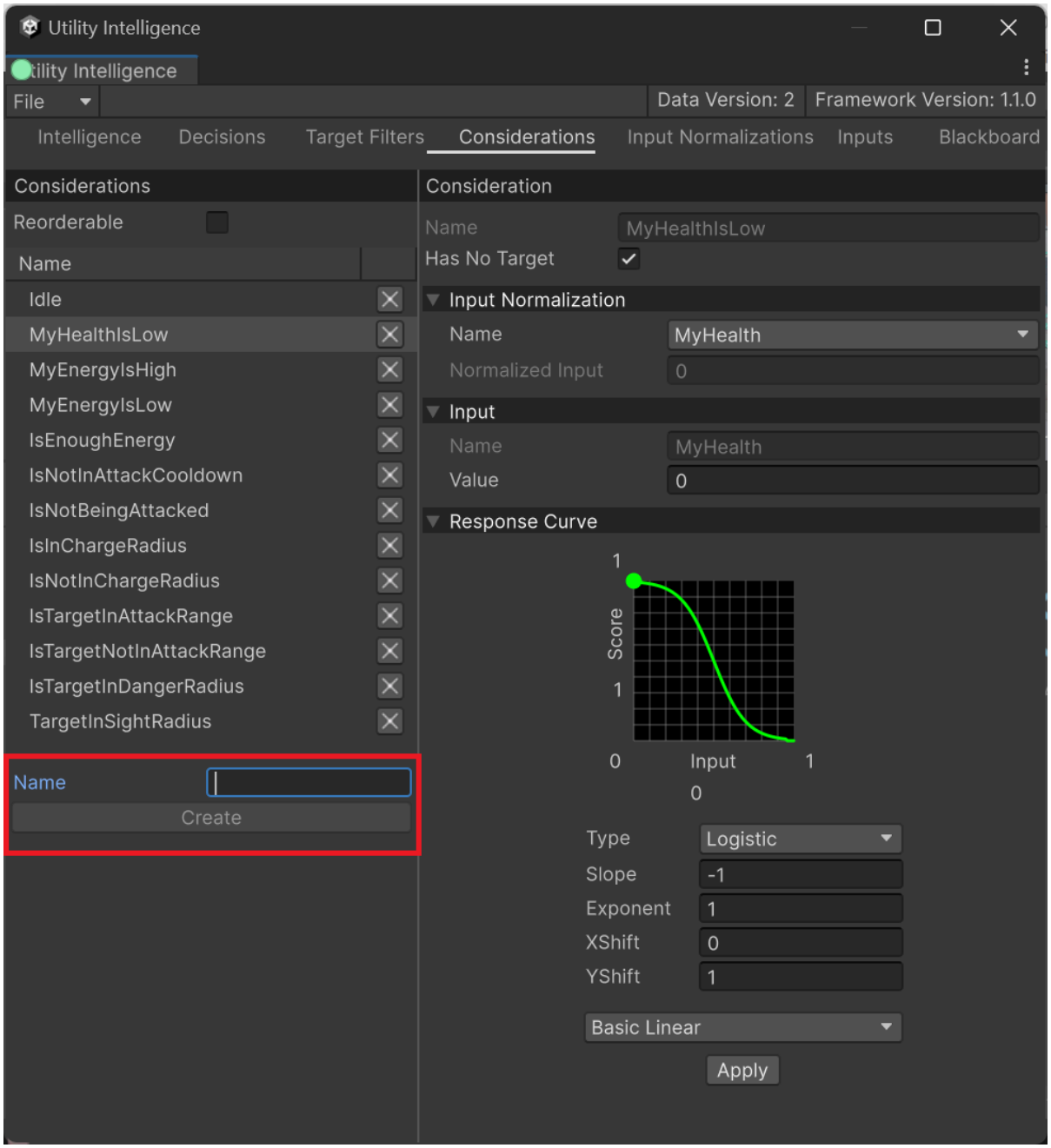
```
public static float CompensateScore(float considerationScore, float considerationCount)
{
    float modificationFactor = 1.0f - 1.0f / considerationCount;
    float makeUpValue = (1.0f - considerationScore) * modificationFactor;
    return considerationScore + makeUpValue * considerationScore;
}
```

To enable/disable Compensation Factor, you need to check/uncheck the Compensation Factor option in the **Intelligence Editor**.



Creating Considerations

To create a new consideration, you need to go to the **Consideration Tab**, fill in the **Name** field, and then click the **Create** button:



After creating a consideration, you can select an Input Normalizations for it, and adjust the Response Curve and observe how these changes affect the consideration score.

Consideration Statuses

Considerations only have two statuses at both runtime and editor time:

: Executed

: Discarded

Response Curves

After an input is normalized into `[0, 1]` by an input normalization, we need a way to map the normalized input to a consideration score that indicates **how appealing** the decision is based on the consideration aspect. This is the role of response curves.

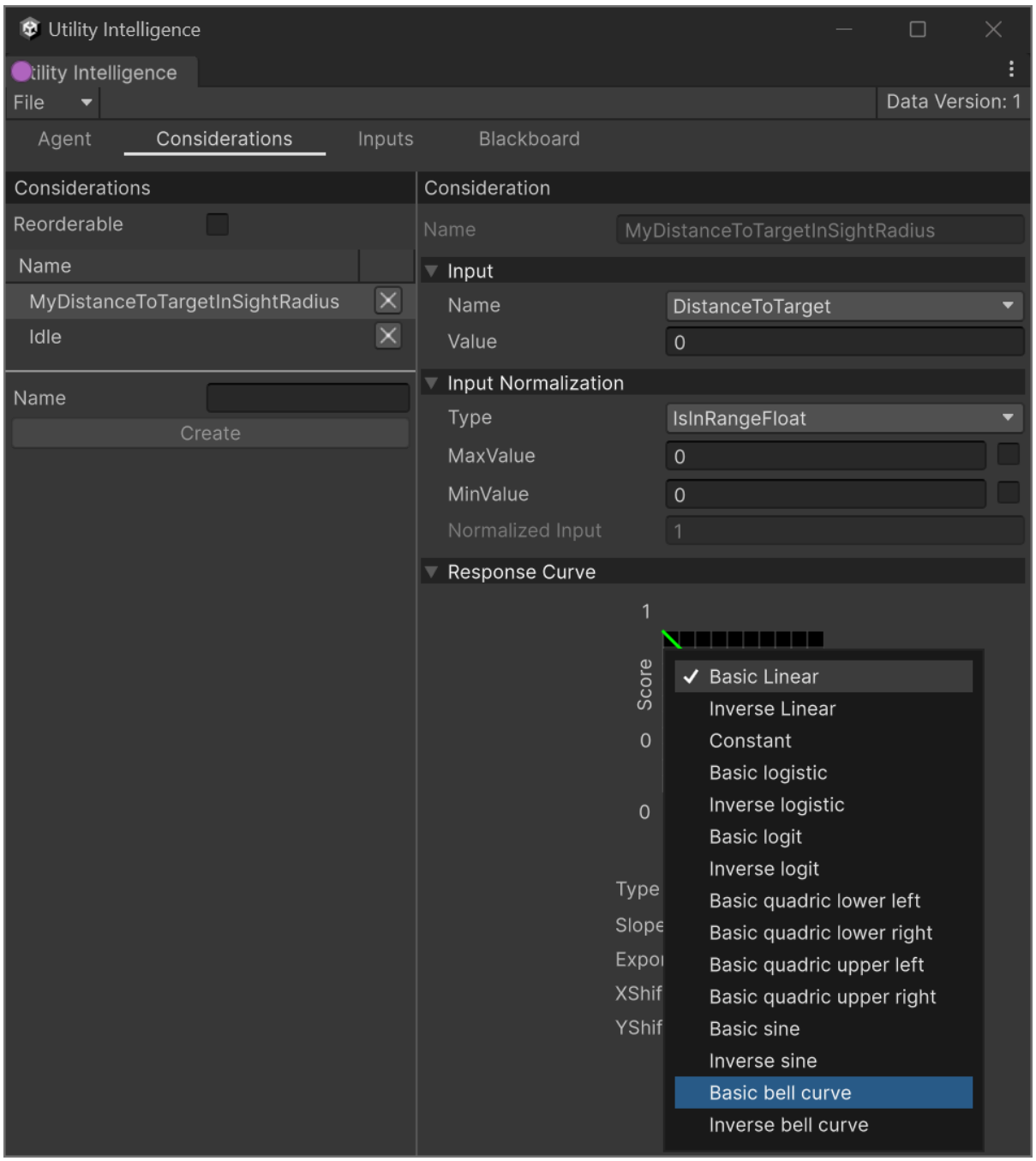
In the previous example, the consideraton **EnemyHealthIsLow** has a response curve that linearly inverts the normalized enemy's health. This curve returns the consideration score that indicates **how appealing** the decision is based on the **enemy's health aspect**. Therefore, the higher the enemy's health, the lower the appeal of the decision.

A response curve has 5 parameters:

- Curve Type
- Slope
- Exponent
- XShift
- YShift

You can change these parameters to adjust the shape of the response curve based on your needs.

Utility Intelligence also provides a list of useful presets for response curves. If you want to use our presets, you just need to select one and click the **Apply** button.



Tip

You can adjust the input values and response curves in the Consideration Tab to observe how they affect the consideration scores.

Inputs

An **input** is some knowledge about the game world that is used to calculate the score of a consideration. For example:

- My health
- Enemy's health
- Distance to the enemy

Creating Inputs

There are two ways to create a new input:

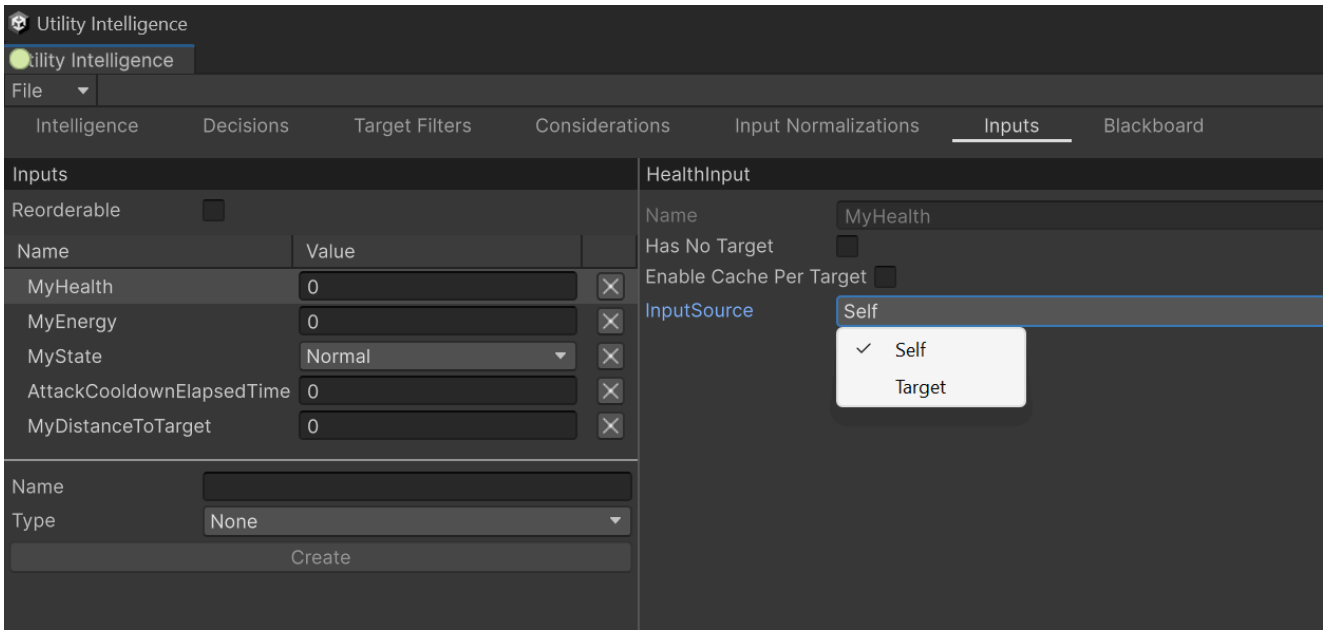
1. Define a new class that inherits from `Input<TValue>` and override the `OnGetRawInput` funnction. For example:

```
public class DistanceToTargetInput : Input<float>
{
    protected override float OnGetRawInput(in InputContext context)
    {
        var currentPos = AgentFacade.Position;
        var targetPos = context.TargetFacade.Position;
        currentPos.Y = 0;
        targetPos.Y = 0;

        return Vector3.Distance(currentPos, targetPos);
    }
}
```

2. Define a new class that inherits from `InputFromSource<TValue>` and override the `OnGetRawInput` function.

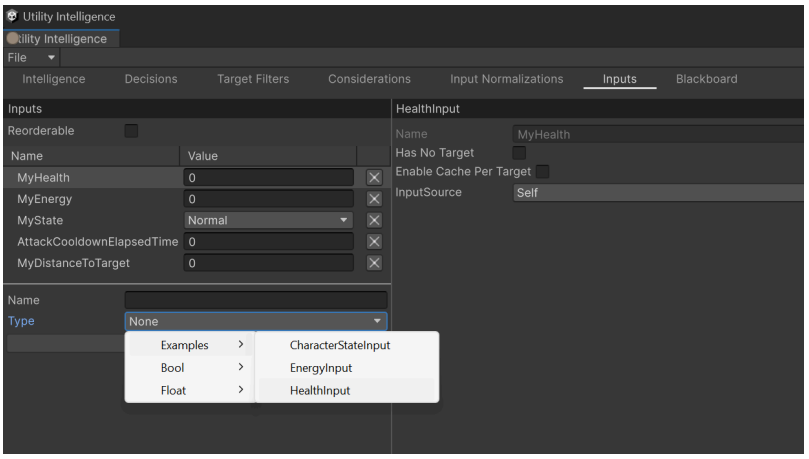
```
[Category("Examples")]
public class HealthInput : InputFromSource<int>
{
    protected override int OnGetRawInput(in InputContext context)
    {
        UtilityEntity inputSource = GetInputSource(in context);
        if (inputSource.EntityFacade is Character character)
        {
            return character.Health;
        }

        return 0;
    }
}
```

- This method allows you to set the source of the input to **Self** or **Target**.

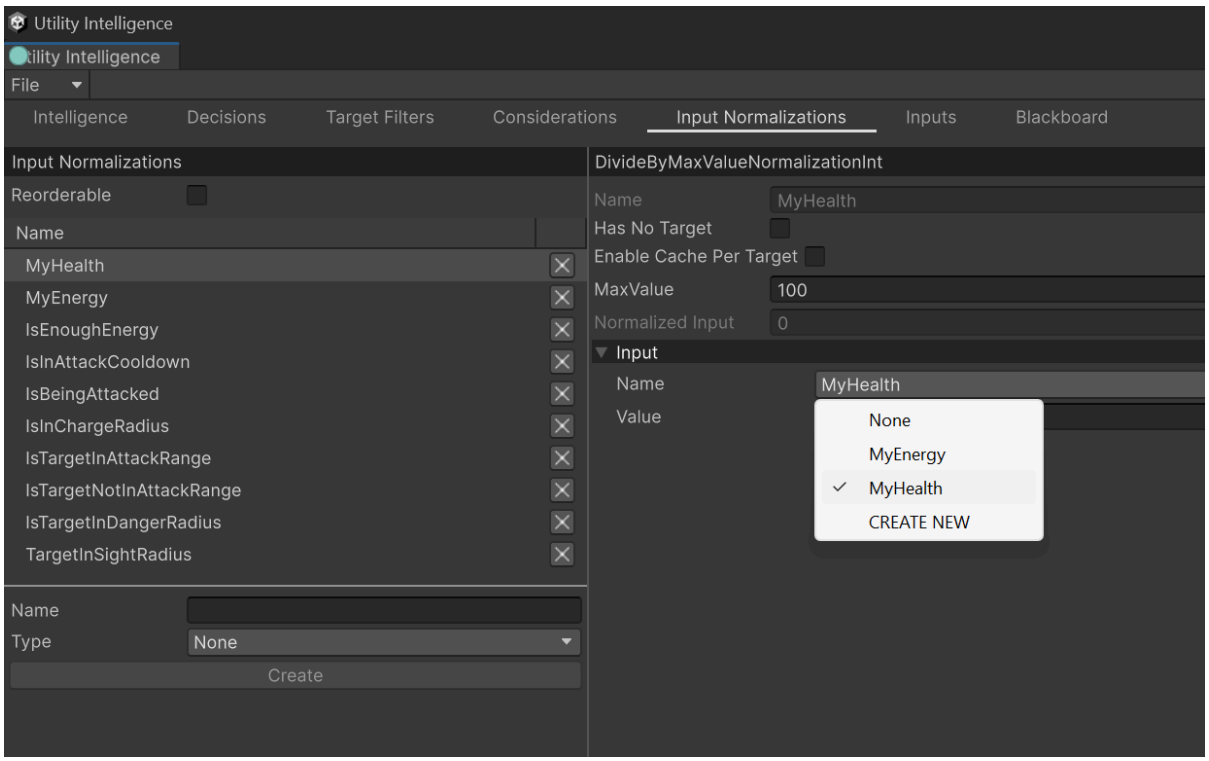


- Note: Use this method only if the input exists in both **Self** and **Target**.

To add the input to the intelligence asset, go to the **Input Tab**, select the input type, give it a name, and then click the **Create** button:



To attach an input to an input normalization, select the input normalization in the **Input Normalization Tab**, and then choose the input's name from the dropdown menu:



Note

Note: Only inputs with the same value type as the input normalization can be attached to it.

Tip

You can adjust the input values in the Intelligence Tab to observe how these changes affect the statuses of considerations and decisions. For further details, check Status Preview.

Supported Value Types

Currently, only the supported value types can be adjusted using the Utility Intelligence Editor to preview which decision is chosen with the Status Preview feature.

Therefore, you should use these types to enable the Status Preview feature. However, you can still use other types if you don't need this feature. For unsupported types, you can only modify the input values by overriding `OnGetRawInput()` function.

Adding Parameter Fields

There are many cases when you need to add parameters to an input to customize its return value. To achieve this, you need to declare these parameters as public fields in your inputs. Here are some examples of how to do this:

```
public abstract class InputFromSource<T> : Input<T>
{
    public InputSource InputSource;

    protected UtilityEntity GetInputSource(in InputContext context)
    {
        if (InputSource == InputSource.Self)
            return Agent;
        if (InputSource == InputSource.Target)
            return context.Target;

        return null;
    }
}
```

```
public abstract class BasicInput<T> : Input<T>
{
    public VariableReference<T> InputValue;
    protected override T OnGetRawInput(in InputContext context)
    {
        return InputValue.Value;
    }
}
```

Supported Field Types

Currently, only the supported field types can be serialized to **JSON** and adjusted using the Utility Intelligence Editor. Therefore, you should use these types when declaring parameter fields for your inputs.

Built-in Inputs

Currently, **Utility Intelligence** provides these buit-in inputs:

- **BasicInputFloatInt**
- **BasicInputBool**
- **BasicInputFloat**
- **BasicInputDouble**
- **BasicInputLong**
- **BasicInputVector2**
- **BasicInputVector3**
- **BasicInputVector2Int**
- **BasicInputVector3Int**
  - Returns the value from its **InputValue** field, which can reference a variable in the Blackboard.
- **DistanceToTargetInput**: Returns the distance from the current agent to the target.

- **CooldownElapsedTimeInput**: Returns the elapsed time since the cooldown started.
- **RaycastToTargetInput**: Returns true if the raycast hits the target; otherwise, returns false.

Input Normalizations

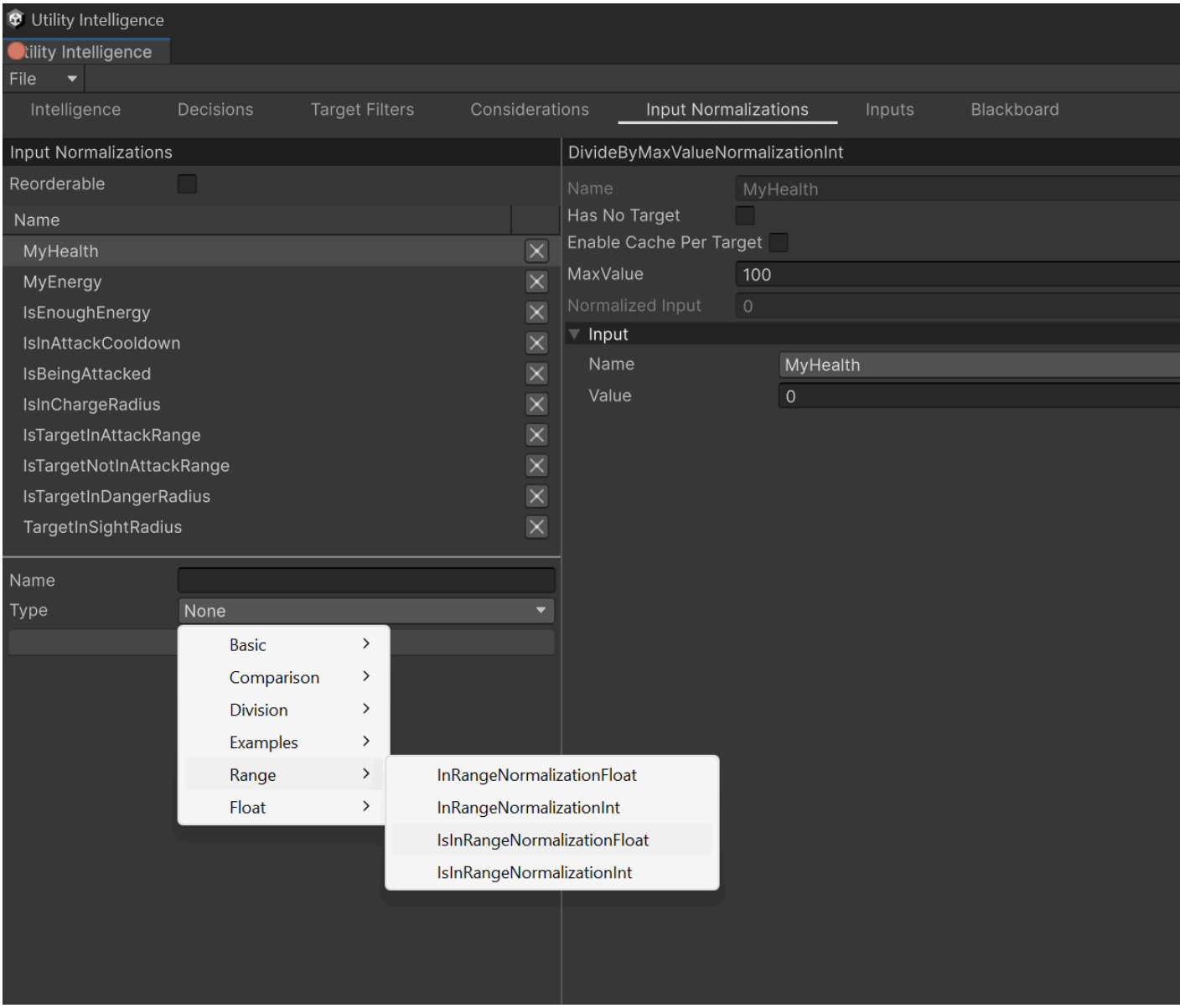
Since an input can vary widely in value, so we need a tool to normalize it into a fixed range `[0,1]`. This is where **input normalizations** come into play. This step is crucial because it ensures that decisions are scored on a consistent scale, allowing us to compare their scores and select the highest-scoring decision.

Creating Input Normalizations

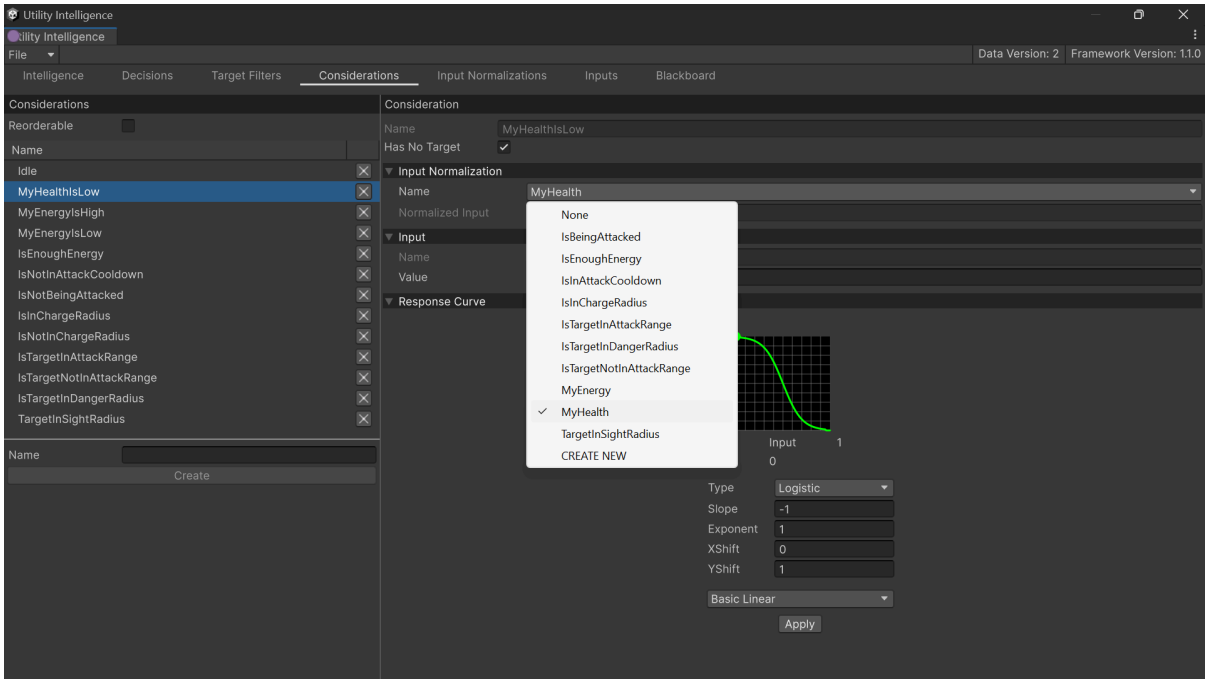
To create a new input normalization, define a new class that inherits from `InputNormalization<TValue>` and override the `OnCalculateNormalizedInput` method. For example:

```
[Category("Range")]
public class IsInRangeNormalizationFloat : InRangeNormalization<float>
{
    protected override float OnCalculateNormalizedInput(float rawInput, in InputNormalizationContext context)
    {
        float normalizedInput = rawInput >= MinValue && rawInput <= MaxValue ? 1.0f : 0.0f;
        return normalizedInput;
    }
}
```

To add the input normalization to the intelligence asset, go to the **Input Normalization Tab**, select the input normalization type, give it a name, and then click the **Create** button:



To attach an input normalization to a consideration, select the consideration in the **Consideration Tab**, and then choose the input normalization's name from the dropdown menu:



Note

Note: Input normalizations can only accept inputs with the same value type.

Supported Value Types

Currently, only the supported value types can be adjusted using the Utility Intelligence Editor. Additionally, inputs can only be attached to input normalizations if they share the same value type. Therefore, you should use these types to enable the Status Preview feature to preview which decision is chosen by modifying the input values in the **Intelligence Editor**. However, you can still use other types if you don't need this feature.

Adding Parameter Fields

There are many cases when you need to add parameters to an input normalization to customize how it normalizes its input value. To achieve this, you need to declare these parameters as public fields in your input normalizations. Here are some examples of how to do this:

```
public class IsInCooldownNormalization : InputNormalization<float>
{
    public VariableReference<float> CooldownDuration;

    protected override float OnCalculateNormalizedInput(float rawInput, in InputNormalizationContext context)
    {
        if (rawInput <= CooldownDuration)
            return 1.0f;
        else
            return 0.0f;
    }
}
```

```
public abstract class InRangeNormalization<TValue> : InputNormalization<TValue>
{
    public VariableReference<TValue> MinValue;
    public VariableReference<TValue> MaxValue;
}

[Category("Range")]
public class InRangeNormalizationFloat : InRangeNormalization<float>
{
    protected override float OnCalculateNormalizedInput(float rawInput, in InputNormalizationContext context)
    {
        var diff = MaxValue - MinValue;
        if (diff <= 0.0f) return 0.0f;

        float normalizedInput = (rawInput - MinValue) / (diff);
        return normalizedInput;
    }
}

[Category("Range")]
public class InRangeNormalizationInt : InRangeNormalization<int>
{
    protected override float OnCalculateNormalizedInput(int rawInput, in InputNormalizationContext context)
    {
        var diff = MaxValue - MinValue;
        if (diff <= 0) return 0.0f;

        float normalizedInput = (float)(rawInput - MinValue) / (diff);
        return normalizedInput;
    }
}
```

Supported Parameter Types

Currently, only the supported field types can be serialized to **JSON** and adjusted using the Utility Intelligence Editor. Therefore, you should use these types when declaring parameter fields for your input normalizations.

Built-in Input Normalizations

We provides a lot of built-in input normalizations to help you normalize your inputs **without having to write a single line of code**:

- Float
  - **BasicNormalizationFloat**: Clamps the input value into `[0, 1]`
  - **DivideByMaxValueFloat**: Divides the input by `MaxValue`.
  - **GreaterThanOrEqualToValueFloat**: Returns `1` if the input value is greater than `Value`; otherwise, returns `0`.
  - **LessThanOrEqualToValueFloat**: Returns `1` if the input value is less than the `Value`; otherwise, returns `0`.

- **InRangeFloat**: Maps the input value from `[MinValue, MaxValue]` to `[0, 1]`. Note that if the input value is above `MaxValue`, then the normalized value is `1`, and if the input value is below `MaxValue`, then the normalized value is `0`.
- **IsInRangeFloat**: Returns `1` if the input value is in the range `[MinValue, MaValue]`; otherwise, returns `0`.
- **IsInCooldownNormalization**: Returns `1` if the input (CooldownElapsedTimeInput) is within the cooldown duration; otherwise, returns `0`.

- Int
  - Similar to the floats
- Bool
  - **BasicNormalizationBool**: Returns `1` if the input value is `true`; otherwise, returns `0`.

Blackboard

Blackboard is used to share information between multiple components in an Agent.

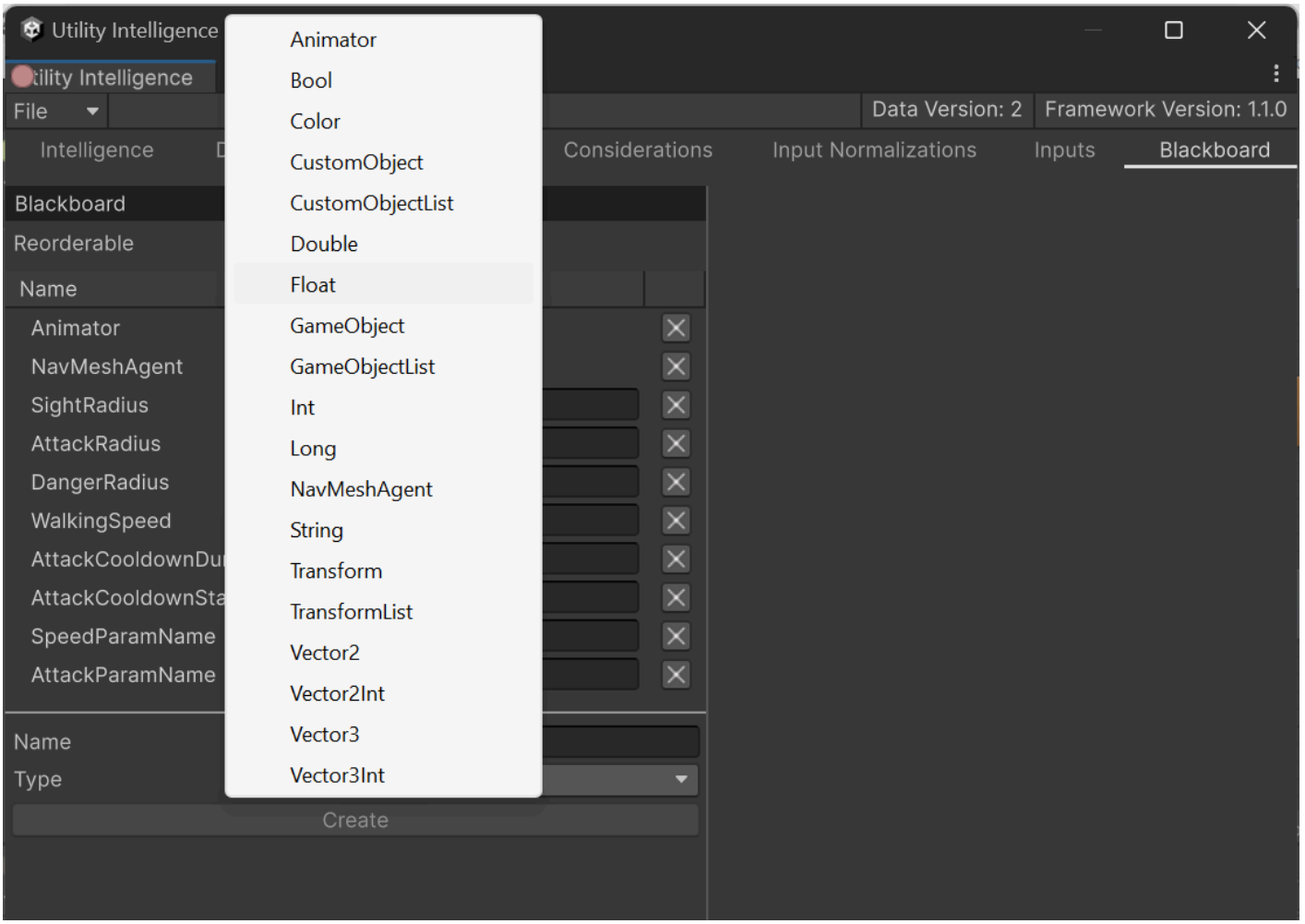
- It can be access from many places, such as Inputs, Input Normalizations, Target Filters, Actions.
- It contains a list of variables and you can Read/Write to these variables for any purpose.

Creating Variables

To create a new variable, define a new class that inherits from `Variable<TValue>`. For example:

```
public class FloatVariable : Variable<float>
{
    public static implicit operator FloatVariable(float value)
    {
        return new FloatVariable { Value = value };
    }
}
```

To add the variable to the intelligence asset, go to the **Blackboard Tab**, select the variable type, give it a name, and then click the **Create** button:



Supported Value Types

Currently, only the supported value types can be serialized to **JSON** and adjusted using the Intelligence Editor.

Therefore, you should use these types for your Blackboard Variables. However, you can still use other types if you don't need to serialize them to JSON. For unsupported types, you need to add them to the Blackboard at runtime like this:

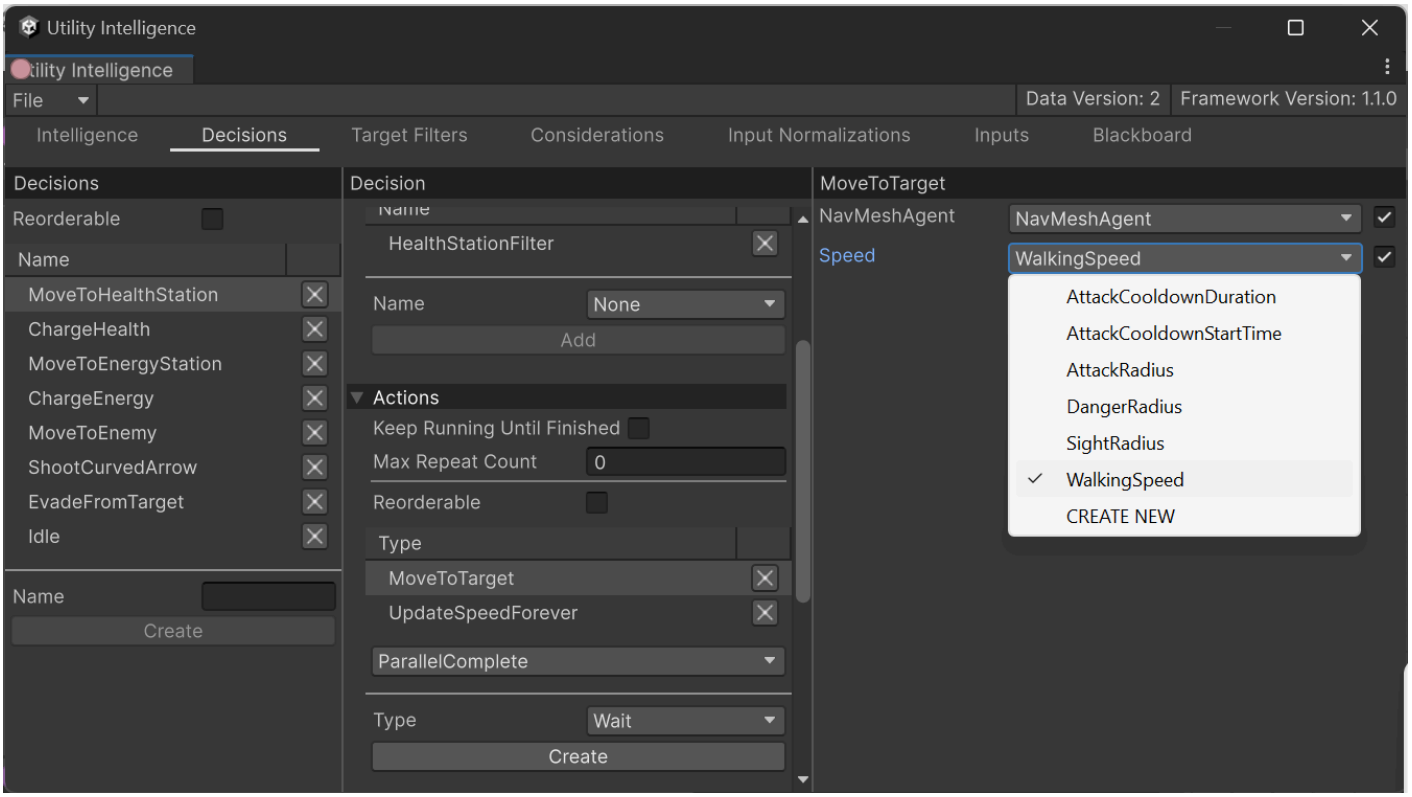
```
public class PatrolWaypoints : MonoBehaviour
{
    public List<Transform> Waypoints;
    private void Start()
    {        Character character = GetComponent<Character>();
        var blackboard = character.Entity.Intelligence.Blackboard;
        var waypointsVariable = blackboard.GetVariable<TransformListVariable>(BlackboardVariableNames.Waypoints);
        waypointsVariable.Value = Waypoints;
    }}
```

Referencing Variables

To reference the variable from an action task, declare a public field of type `VariableReference<TValue>` in the action task's class. For example:

```
[Category("NavMeshAgent")]
public class MoveToTarget : NavMeshActionTask
{
    public VariableReference<float> Speed = 5;

    protected override void OnStart()
    {
        navMeshAgent.speed = Speed;
        MoveToTarget();
    }

    protected override UpdateStatus OnUpdate(float deltaTime)
    {
        if (HasArrived())
            return UpdateStatus.Success;

        return UpdateStatus.Running;
    }

    protected override void OnEnd()
    {
        StopMove();
    }
}
```

Then select the action task in the **Decision Tab** and choose the variable's name from this dropdown menu:



Built-in Variables

Currently, we provides these built-in variables:

- Float
- Double
- Int
- Long
- Bool
- String
- Vector2
- Vector2Int
- Vector3
- Vector3Int
- Color
- GameObject
- GameObjectList
- Transform
- TransformList
- Animator
- NavMeshAgent
- ScriptableObject
- ScriptableObjectList

Tips & Tricks

Tips & Tricks

General Tips & Tricks

Ask AI ChatBots

We provided detailed documentation in **PDF** format. So, if you need instant answers to your questions, upload our documentation to the AI ChatBots, such as **ChatGPT**, **Claude,** or **Perplexity**, and then you can ask about anything you don't understand regarding **Utility Intelligence**. It's much easier for beginners to learn **Utility Intelligence** this way.

Use GitHub Copilot

GitHub Copilot recently released a free plan. So, if you are having trouble getting started because you don't understand our code in the examples, you can ask **GitHub Copilot** to explain it to you for free.

Other Tips & Tricks

1. **Intelligence Editor**
2. **Considerations**
3. **Decisions**
4. **Decision Makers**
5. **Utility Worlds**

Intelligence Editor

Use Status Preview

After making changes to considerations or decisions, you can use our **Status Preview** feature to check whether the results are as you expected.

- For better results, if your inputs have targets, you should create a separate input for each kind of target so that you can change the input value for each target type to see how it affects the decision scores. For example, you can create `DistanceToEnemyA`, `DistanceToEnemyB`, `DistanceToHealthStation`, `DistanceToEnergyStation`, etc., adjust their values, and then check the result in the **Intelligence Editor**.

Lock the Intelligence Editor

If you want to modify variables from other **GameObjects** and see how they affect the decision scores, you can lock the **Intelligence Editor**, select other **GameObjects**, change those variables in the **Inspector Window**, and see the results in the **Intelligence Editor**.

- For example, suppose you have an input called **DistanceToTarget**, and you want to see how it affects the decision scores at runtime. You can lock the **Intelligence Editor**, drag the target around the current scene, and then check the result in the **Intelligence Editor**.

For more information, please read: **Lock the Editor**

Group your components into categories

As your AI system becomes more complex, you will have many inputs, input normalizations, considerations, and decisions, making it challenging to manage. You should group them into categories for easier management by using **CategoryAttribute** and **CategoryField**.

For more information, please check: **Categories**

Organize Fields in the IntelligenceEditor

After your classes become more complex and have a lot of fields, you can organize your fields in the **Intelligence Editor** by using the **Field Attributes**

Change class names and field names in JSON

When you change the class names or field names of a serializable component (Input, InputNormalization, TargetFilter, or ActionTask), you can use the **JSON Attributes** to change those names in **JSON**.

Considerations

There are 3 kinds of considerations:

- Boolean consideration: Returns a score of `0.0` or `1.0`.
  - Use this if you need to consider a **Yes-No** question. If the answer is `yes`, it returns `1.0`, and vice versa.
  - For example:
    - IsTargetInAttackRange
    - IsTargetInDamageArea
    - IsInAttackCooldown
- Variable consideration: Returns a score from `0.0` to `0.1`.
  - Use this if you need a consideration that returns a score that changes dynamically based on the current input.
  - For example:
    - TargetInSightRadius
    - TargetInAttackRange
    - MyHealthIsLow
    - TargetHealthIsHigh
- Constant consideration: Returns a constant score in `[0.0, 1.0]`. Use this when:
  - You need a consideration that always returns a constant score, e.g., `0.1` or `0.2`, etc.
  - You need a fallback decision that will be selected if the the agent doesn't know which decision to choose in the current situation.
  - For example:
    - Idle

Common Consideration Recipes

Distance

IsTarget(Not)InRange

- Returns `1.0` if the target is within the specified range or `0.0` if it is outside the range.
- Recipe
  - Input: `DistanceToTargetInput`
  - InputNormalization: `IsInRangeNormalization`

- ResponseCurve: `Basic Linear` (`Inverse Linear`)

TargetInRange

- Returns a score in `[0.0, 1.0]`. It maps the input value (`DistanceToTargetInput`) from `[Start, End]` to `[0.0, 1.0]`.
  - If the input value is less than **Start**, returns `0.0`
  - If the input value is greater than **End**, returns `1.0`
- Recipe
  - Input: `DistanceToTargetInput`
  - InputNormalization: `InRangeNormalization`
  - ResponseCurve: **Which ResponseCurve should you use?**

Is(StateName)State

- Returns `1.0` if input state is the specified state; otherwise returns `0.0`.
- Recipe:
  - Input: User Custom Input (often an `Enum`)
    - Returns a state of the agent or the target.
  - InputNormalization: User Custom Input
    - Returns `1.0` if input state is the specified state.
    - Returns `1.0` if input state is not the specified state.
  - ResponseCurve: `Basic Linear` (`Inverse Linear`)

Is(Not)InCooldown

- Returns `1.0` if the `CooldownElapsedTimeInput` is within the cooldown duration; otherwise returns `0.0`
- Recipe:
  - Input: `CooldownElapsedTimeInput`
  - InputNormalization: `IsInCooldownNormalization`
  - ResponseCurve: `Basic Linear` (`Inverse Linear`)

Random

- Returns a random score in `[0.0, 1.0]`
- Recipe:
  - Input: User Custom Input
    - Returns a random input value.

- InputNormalization: `BasicNormalization`
- ResponseCurve: `Basic Linear`

Health

- The input value will be normalized by dividing by `100`
  - If the input is less than `0.0`, returns `0.0`
  - If the input is greater than `100`, returns `1.0`
- Recipe
  - Input: Health
  - InputNormalization: `DivideByMaxValue`
  - ResponseCurve: **Which ResponseCurve should you use?**

Idle

- Returns a constant score (often `0.1`)
- Recipe:
  - Input: None
  - InputNormalization: None
  - ResponseCurve: `Constant`

Which ResponseCurve should you use?

- Boolean considerations:
  - `Basic Linear` or `Inverse Linear`
- Variable considerations:
  Suppose the input gradually increases from `0.0` to `1.0`:
  - The score is proportional to the input and increases gradually.
    - Linear: `Basic Linear`
    - Slow at first, fast later: `Basic Quadric Lower Right`
    - Fast at first, slow later: `Basic Quadric Upper Left`
    - Slow at either end, fast in the middle: `Basic Logistic`
    - Fast at either end, slow in the middle: `Basic Logit`
  - The score is inversely proportional to the input and decreases gradually.
    - Linear: `Inverse Linear`
    - Slow at first, fast later: `Basic Quadric Upper Right`

    - Fast at first, slow later: `Basic Quadric Lower Left`
    - Slow at either end, fast in the middle: `Inverse Logistic`
    - Fast at either end, slow in the middle: `Inverse Logit`
  - The score fluctuates
    - Slow at either end, fast in the middle: `Basic Bell Curve` or `Inverse Bell Curve`
    - Fast at either end, slow in the middle: `Basic Logit` or `Inverse Logit`
    - More dynamic: `Basic Sine` or `Inverse Sine`
- Constant considerations:
  - `Constant`

Decisions

Enable Compensation Factor

If your agents have decisions that contain a lot of considerations, you should enable **Compensation Factor** to ensure the decision scores are not quite low.

For more information about **Compensation Factor**, see: **Compensation Factor**

Enable KeepRunningUntilFinished

If your agents have an important decision that you don't want to interrupt while it is running, regardless of whether there is another decision with a higher score, you can enable the **KeepRunningUntilFinished** option of the decision to prevent the agents from making a new decision while it is running.

For more information about **KeepRunningUntilFinished**, see: **KeepRunningUntilFinished**

Use MomentumBonus

If you want to prioritize the **last chosen** decision-target pair in the **next** decision-making round, you can set the **Momentum Bonus** to a value greater than **1.0** (usually between **1.1** -> **1.25**). In the next decision-making round, the **last chosen** decision-target pair will be prioritized by multiplying its score by the **Momentum Bonus**, increasing its chances of winning and thereby reducing oscillation between nearly equal decision-target pairs.

For more information about **MomentumBonus**, see: **Momentum Bonus**

Add Fallback Decision

You should add a fallback decision with a constant score so that your agents always have a decision to run.

- For example, in our example scenes, we always add the **Idle** decision with a score of **0.1**. Therefore, when our agents find themselves in a situation where they don't know which decision to choose, they will be idle.

Use Decision Weight

If you want to prioritize one decision over another, you can adjust its weight to be higher than the other.
For example:

- Normal Layer's Weight: 1.0
- Combat Layer's Weight: 2.0
- Urgent Layer's Weight: 3.0

For more information about **Decision Weight**, see: **Decision Weight**

Use empty TargetFilter list for Decisions that target all Entities

If the targets of your decision are all entities in the current utility world, you can leave the **TargetFilter** list of the decision **empty**. For decisions that have targets and an empty **TargetFilter** list, the utility world will pass all its entities to the decision.

How Tos

How to enable/disable a decision based on a condition?

Add a boolean consideration that returns `1.0` (true) or `0.0` (false) depending on the condition result.

How to enable/disable decisions based on states

Add a state consideration to each decision. Check the recipe for a state consideration here: **Common Consideration Recipes**

How to add some randomness to a decision?

Add a random consideration to the decision. Check the recipe for a random consideration here: **Common Consideration Recipes**

How to reduce the oscillation of scores between decision-target pairs

For more information about how to reduce the oscillation of scores between decision-target pairs, see: **Oscillation between decision-target pairs**

Decision Makers

Character Transformation

In case you want to transform your character into another with a different set of behaviors, you can create a separate **Decision Maker** for each kind of character. For example, if you want to transform a warrior into an archer when he picks up a bow, you can create one **Decision Maker** for the warrior and another for the archer.

Note: you need to add a boolean consideration (**HasABow**) to all decisions in the **Archer DM** to enable/disable the DM based on the condition of whether the archer has a bow.

Utility Worlds

Create separate worlds for different purposes

Utility Worlds can serve a variety of purposes within a game. For example, you can create one utility world for handling character behaviors during combat (attacking, moving, fleeing, etc.), one utility world for managing character behaviors in daily routines (eating, sleeping, drinking, resting, etc.), and another for controlling character interactions in social scenarios. Each Utility World can focus on a specific aspect of the game, enabling modular and maintainable AI systems.

The key benefit of using multiple worlds is to reduce the cost of decision-making. When your characters have different sets of behaviors (**Decision Makers**), if you put all of them into one **Intelligence Asset**, the cost of decision-making will be high because behaviors in one set may not be used in other sets (e.g. combat behaviors not being used in daily routines).

Benefits

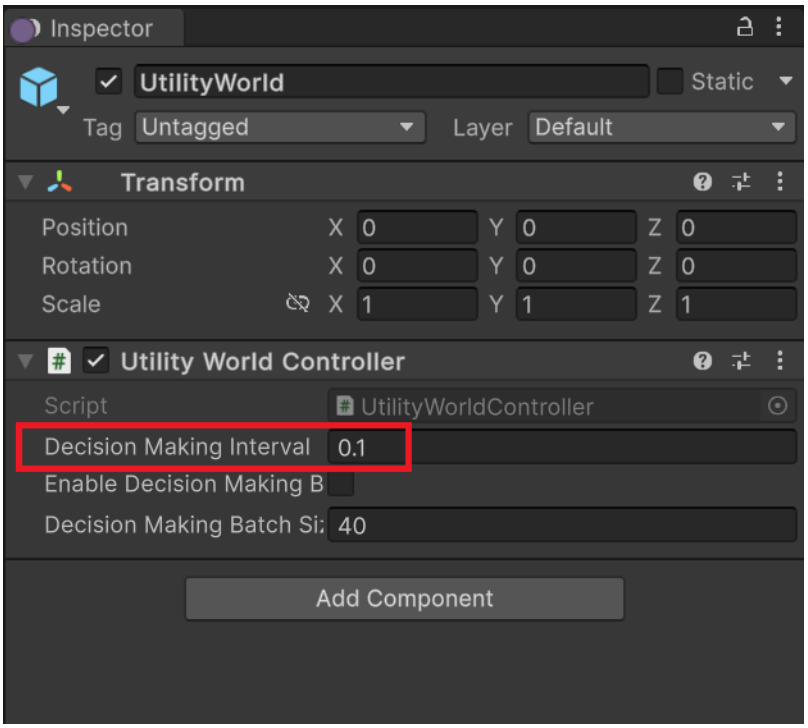
- Reduce the number of considerations, decisions, decision makers that need to be executed.
- Reduce the number of targets that need to be filtered
  - Each decision has different types of targets, so it requires different types of target filters. If you include all decisions in one **Intelligence Asset**, you will need to register all their targets with the utility world. This will increase the cost of filtering targets for these decisions.

Optimization Tricks

Optimizing the decision-making process

Adjust the decision-making interval

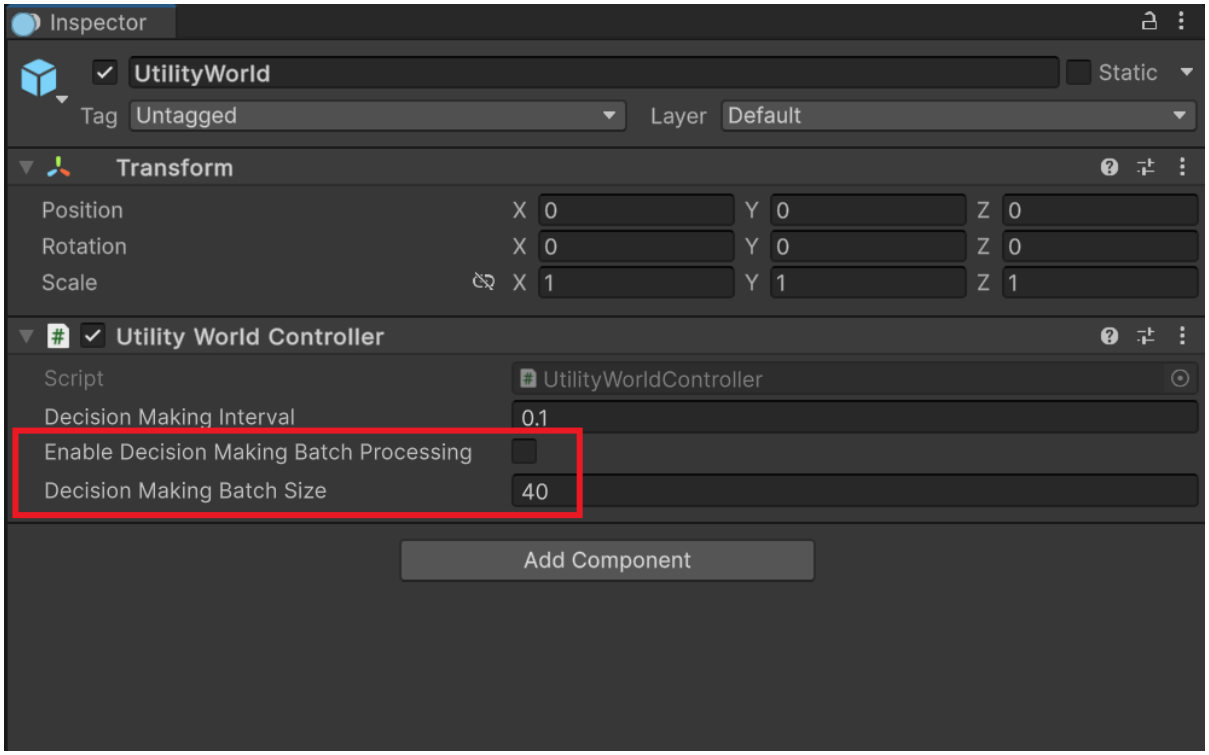
In **Utility Intelligence**, decision-making is separated from decision-execution, allowing you to run decision-making at a different frequency than decision-execution by adjusting the **Decision Making Interval** in the Utility World Controller:



The default decision-making interval is 0.1s. You can inrease it to **0.2s**, **0.3s**, or **0.5s** depending on your game needs. It will help reduce computational burden on the CPU.

Distribute the decision-making task across multiple frames

Starting from **v2.1.0**, we can distribute the decision-making task across multiple frames to balance the workload by checking **Enable Decision Making Batch Processing** in the Utility World Controller.

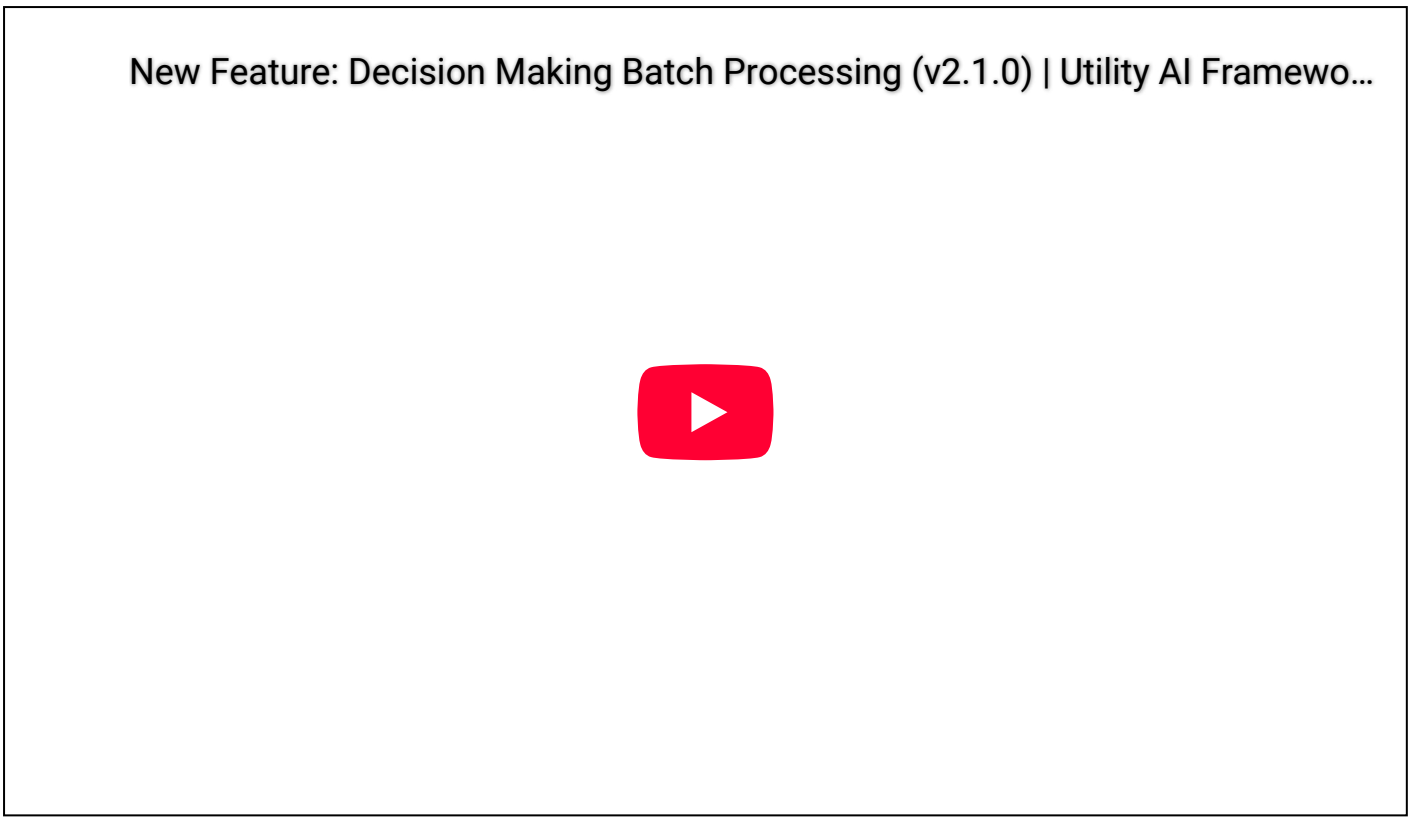


After **Enable Decision Making Batch Processing** is checked, you can set the **Decision Making Batch Size** to limit the number of agents that can make decisions per frame. The default batch size is 40.

For example, if you have 500 agents, and you set the **Decision Making Batch Size** to 20, it will take 25 frames to complete the decision-making process.

This feature will help you handle significantly more agents than before. Previously, the decision-making for all agents in a utility world was processed within a single frame, which could cause spikes in the profiler if you had a high number of agents. Now you can limit the number of agents to 20 per frame, or even to 10 per frame. This will greatly reduce the computational burden per frame on the CPU, and help avoid performance spikes.

Here's my test with 300 agents: the decision-making process runs every 0.25s and processes 10 agents per frame.



Create separate worlds for different purposes

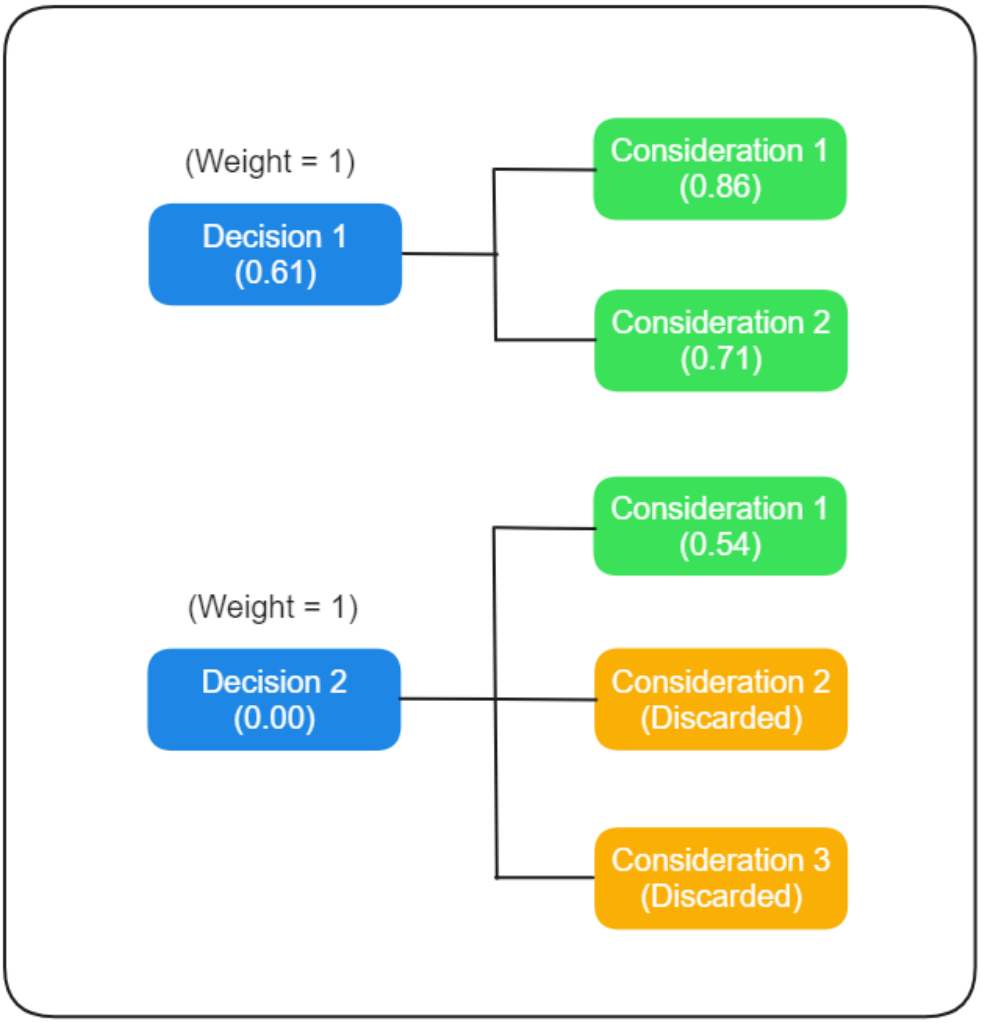
If your agents have different sets of behaviors for different purposes, you should create a separate utility world for each purpose to reduce the cost of decision-making.

For more information, please read: **Why you should create separate worlds for different purposes**.

Optimizing the score-calculation process

Understanding how the process works

Before starting optimization, you need to understand how the **score-calculation process** works first. In Utility Intelligence, the score-calculation process is executed sequentially from top to bottom, and the lower ones are discarded if they cannot possibly beat the higher one.
For example:



In this case, firstly, **Decision 1** is scored, and its final score is `0.61`. This score will be passed into the score-calculation process of **Decision 2** as `minToBeat`.

When calculating the score of **Decision 2**, since its first consideration is scored as `0.54` and the decision weight is `1`, the maximum score of **Decision 2** is `0.54`. Since it is lower than `minToBeat`, **Decision 2** realizes that it cannot beat **Decision 1**. Consequently, all lower considerations are discarded and the final score of **Decision 2** is `0.00`.

For decision makers, they are similar to decisions, if the lower ones realize that they cannot possibly beat the higher one, then they will be discarded, and their final score will be `0.00`.

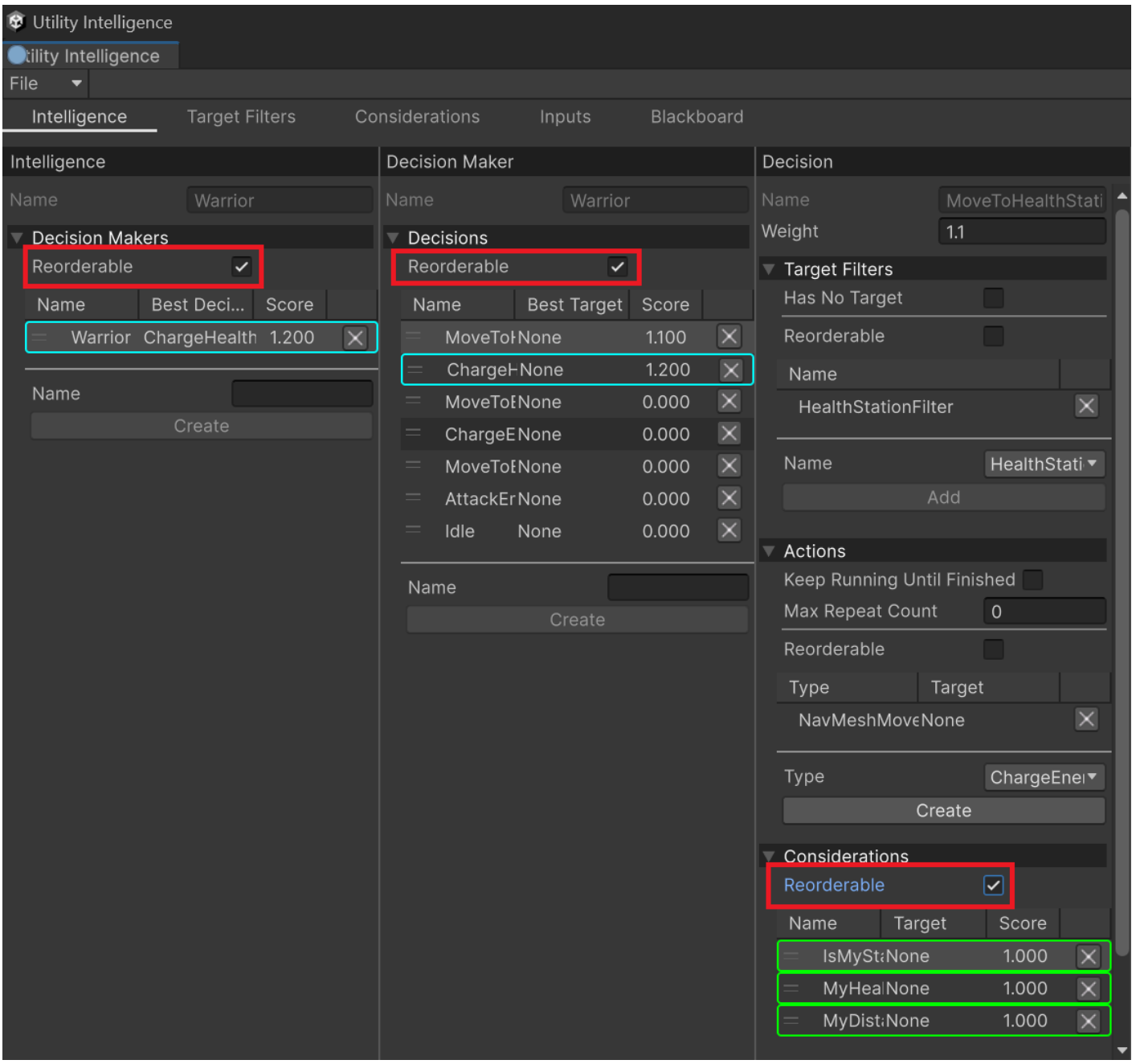
How to optimize the process

Now that you understand how the score-calculation process works, and to optimize this process, follow these guidelines:

Reordering decision makers, decisions, and considerations

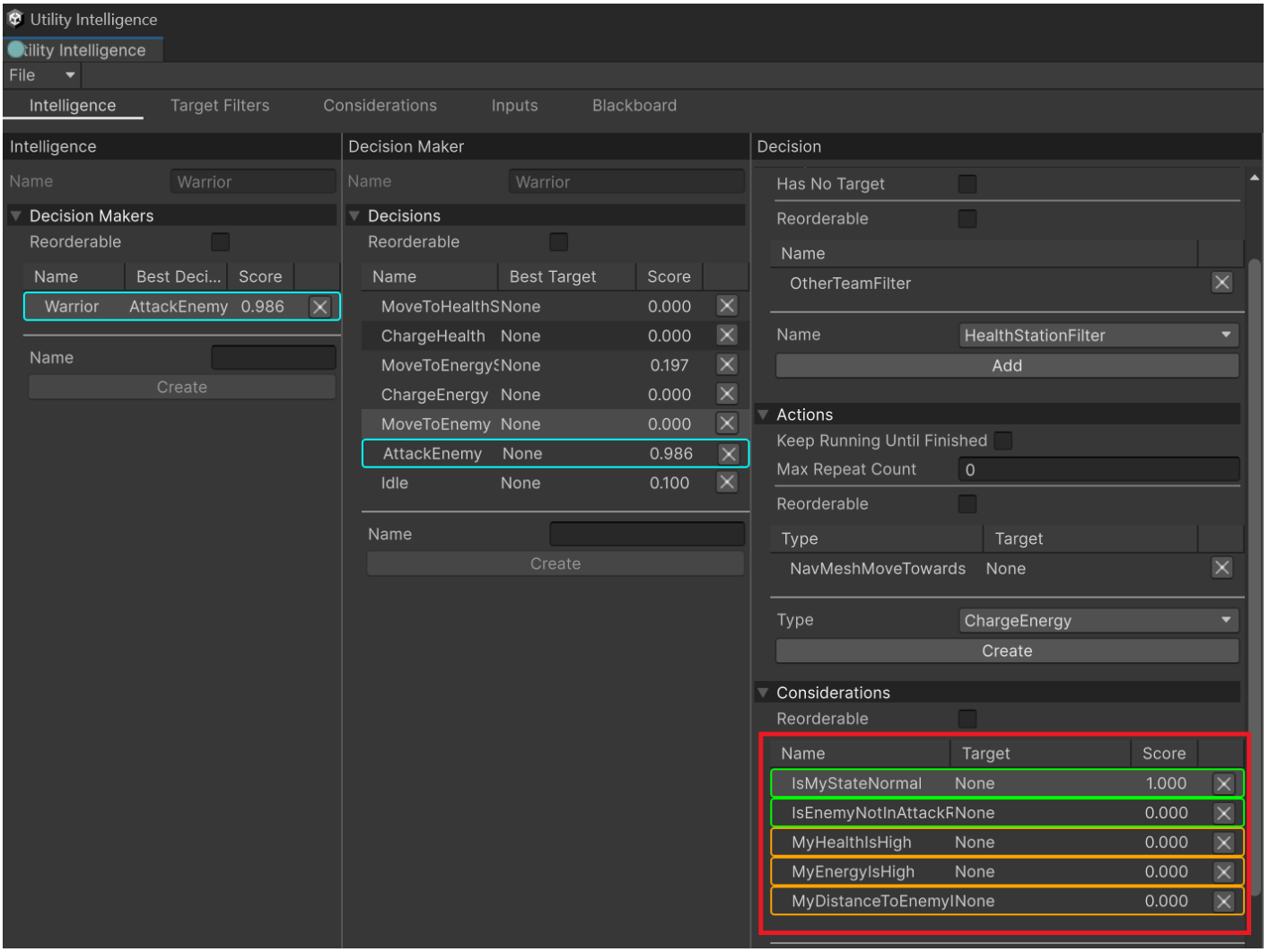
- **Considerations**
  - Put considerations that have a high probability of returning a low score at the top.
    - This ensures that lower considerations will be discarded because it's very difficult for lower decisions to beat the higher ones if their first consideration returns a low score.
    - A good question we should ask ourselves when doing this is: Does this consideration return a low score most of the time? For example:
      - `IsTargetInAttackRange` (it usually returns `0.0` because most of the time the target is not in the attack range).
  - Put considerations that are expensive at the bottom. For example:
    - Considerations using raycasts.
- **Decisions**
  - Put decisions that have a high probability of returning a high score at the top.
    - This ensures that lower decisions will be discarded because it's very difficult for them to beat the higher ones with a high score.
    - A good question we should ask ourselves when doing this is: Does this decision return a high score most of the time? For example:
      - `FindPlayer` (it usually returns high score because most of the monsters are constantly finding the player).
      - Decisions with high weights.
- **Decision Makers**
  - Similar to decisions.

To reorder decision makers, decisions, and considerations, you need to enable the **Reorderable** option in the Editor. This option adds drag handles before every item, allowing you to change the order of each item by dragging it.



Note

- Considerations that are **green** have been executed.
- Considerations that are **orange** have been discarded.
- For more information about the statuses of considerations, check Consideration Statuses



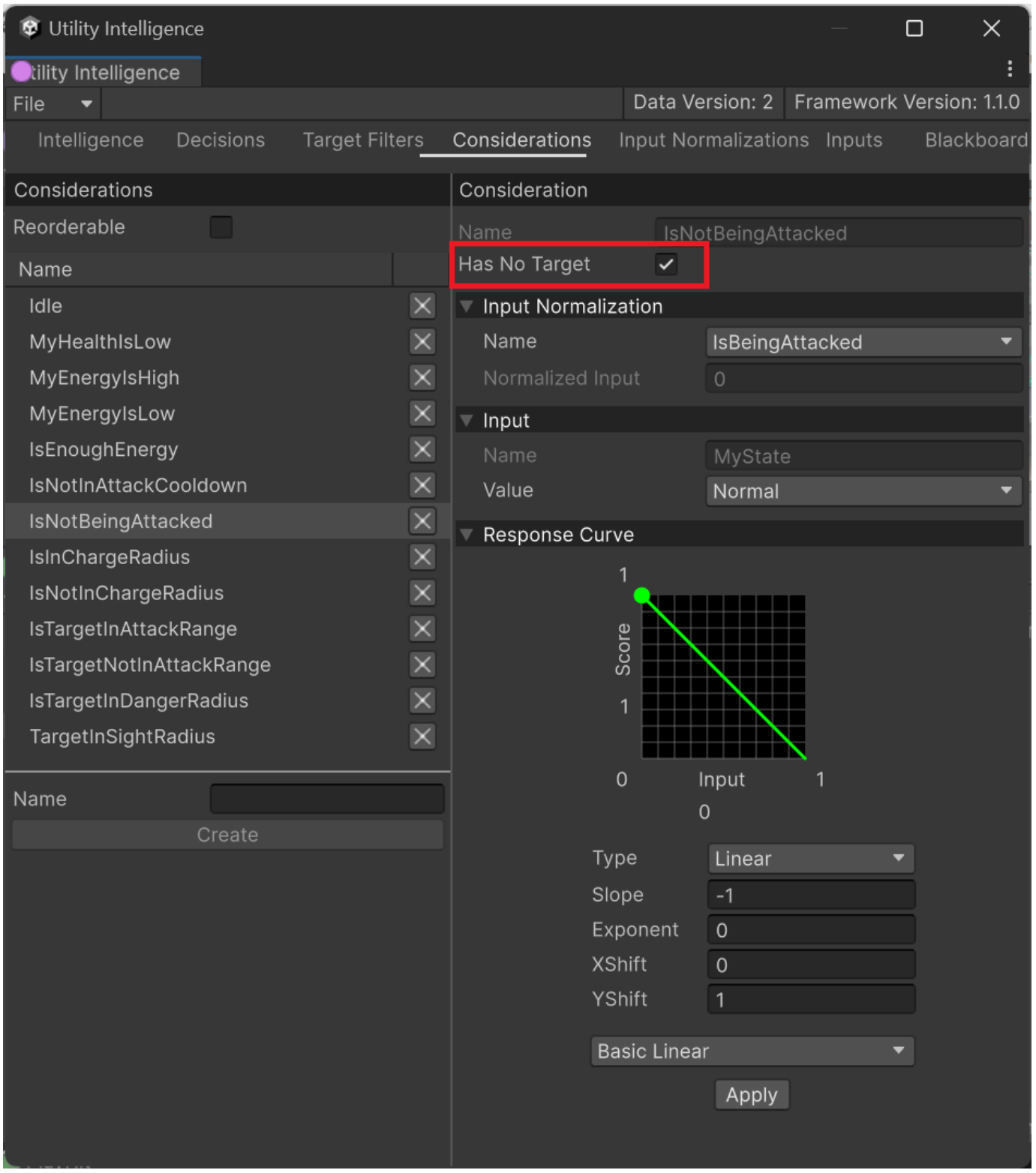
Caching calculated results

Did you know that calculated results from inputs, input normalizations, considerations and decisions can be cached and reused across parent components, thereby eliminating unnecessary recalculations.

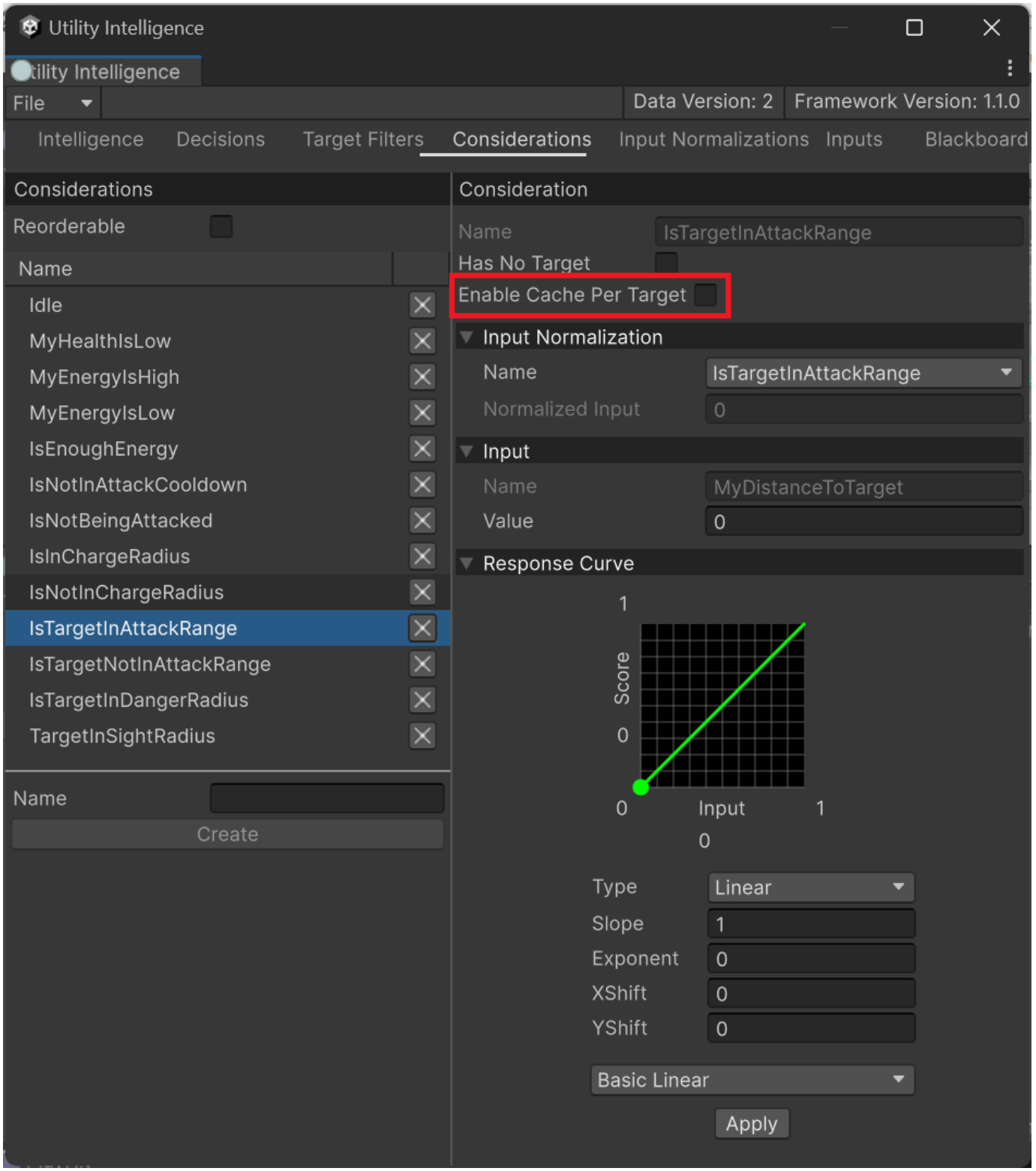
Considerations

To enable caching the calculated score of a consideration:

- If the consideration has no target, check the **HasNoTarget** toggle:



- If the consideration has targets, uncheck the HasNoTarget toggle and check the **EnableCachePerTarget** toggle:


Utility Intelligence
tility Intelligence
File
Data Version: 2
Framework Version: 1.1.0
Intelligence
Decisions
Target Filters
Considerations
Input Normalizations
Inputs
Blackboard
Considerations
Reorderable
Name
Idle
MyHealthIsLow
MyEnergyIsHigh
MyEnergyIsLow
IsEnoughEnergy
IsNotInAttackCooldown
IsNotBeingAttacked
IsInChargeRadius
IsNotInChargeRadius
IsTargetInAttackRange
IsTargetNotInAttackRange
IsTargetInDangerRadius
TargetInSightRadius
Name
Create
Consideration
Name
IsTargetInAttackRange
Has No Target
Enable Cache Per Target
Input Normalization
Name
IsTargetInAttackRange
Normalized Input
0
Input
Name
MyDistanceToTarget
Value
0
Response Curve
1
Score
0
0
Input
1
0
Type
Linear
Slope
1
Exponent
0
XShift
0
YShift
0
Basic Linear
Apply

Note

- **Enable Cache Per Target**
  - Managing scores for individual targets incurs a cost. Therefore, caching is only effective if the cost of caching is lower than the cost of recalculating the score.
  - You should enable caching per target only if the consideration contains heavy inputs, input normalizations and is used by multiple decisions.
- **Has No Target**
  - The consideration is treated as **having no target**. In this case, the consideration score is cached directly within the consideration, eliminating the need to manage scores for individual targets. This results in a very low caching cost.
  - You should enable this option for every consideration that does not access the decision's target.

Inputs, Input Normalizations and Decisions

- Similar to considerations.

Supported Types

Supported Value Types

Currently, only the following value types are supported. You should use these types as value types for Inputs, Input Normalizations, and Blackboard Variables:

- `enum`
- `int`
- `long`
- `float`
- `double`
- `bool`
- `string`
- `Vector2`
- `Vector2Int`
- `Vector3`
- `Vector3Int`
- `Color`

> **Note**
>
> You can still use other types as value types for Inputs, Input Normalizations, and Blackboard Variables. However, they will not be shown in the Intelligence Editor. Therefore, you will not be able to adjust their values through the Intelligence Editor

Supported Field Types

Currently, only the following field types can be serialized to **JSON** and adjusted using the Utility Intelligence Editor. You should use these types when declaring parameter fields for Inputs, Input Normalizations, Action Tasks, and Target Filters.

- `enum`
- `int`
- `long`
- `float`
- `double`

- `bool`
- `string`
- `Vector2`
- `Vector2Int`
- `Vector3`
- `Vector3Int`
- `Color`
- `LayerMask`
- `VariableReference<TValue>`

Attributes

JSON Attributes

Some users have provided feedback that renaming class names and field names is currently quite annoying because it has to be done manually by editing the serialized JSON. So, in v2.2.0, we introduce these attributes to make renaming class names and field names easier and faster.

> **Info**
>
> You can use these attributes for input, input normalizations, action tasks and target filters.

ClassFormerlySerializedAs

To change a class name from `CarlosLab.OldNamespace.OldActionTask` to `CarlosLab.NewNamespace.NewActionTask`, you need to pass the old class name and the old namespace to the constructor of `ClassFormerlySerializedAs`:

```
namespace CarlosLab.NewNamespace
{
    [ClassFormerlySerializedAs(oldClassName:"OldActionTask", oldNamespace:"CarlosLab.OldNamespace")]
    public class NewActionTask : ActionTask
    {
    }
}
```

If the namespace remains unchanged, you only need to pass the old class name:

```
namespace CarlosLab.Unchanged
{
    [ClassFormerlySerializedAs(oldClassName:"OldActionTask")]
    public class NewActionTask : ActionTask
    {
    }
}
```

FieldFormerlySerializedAs

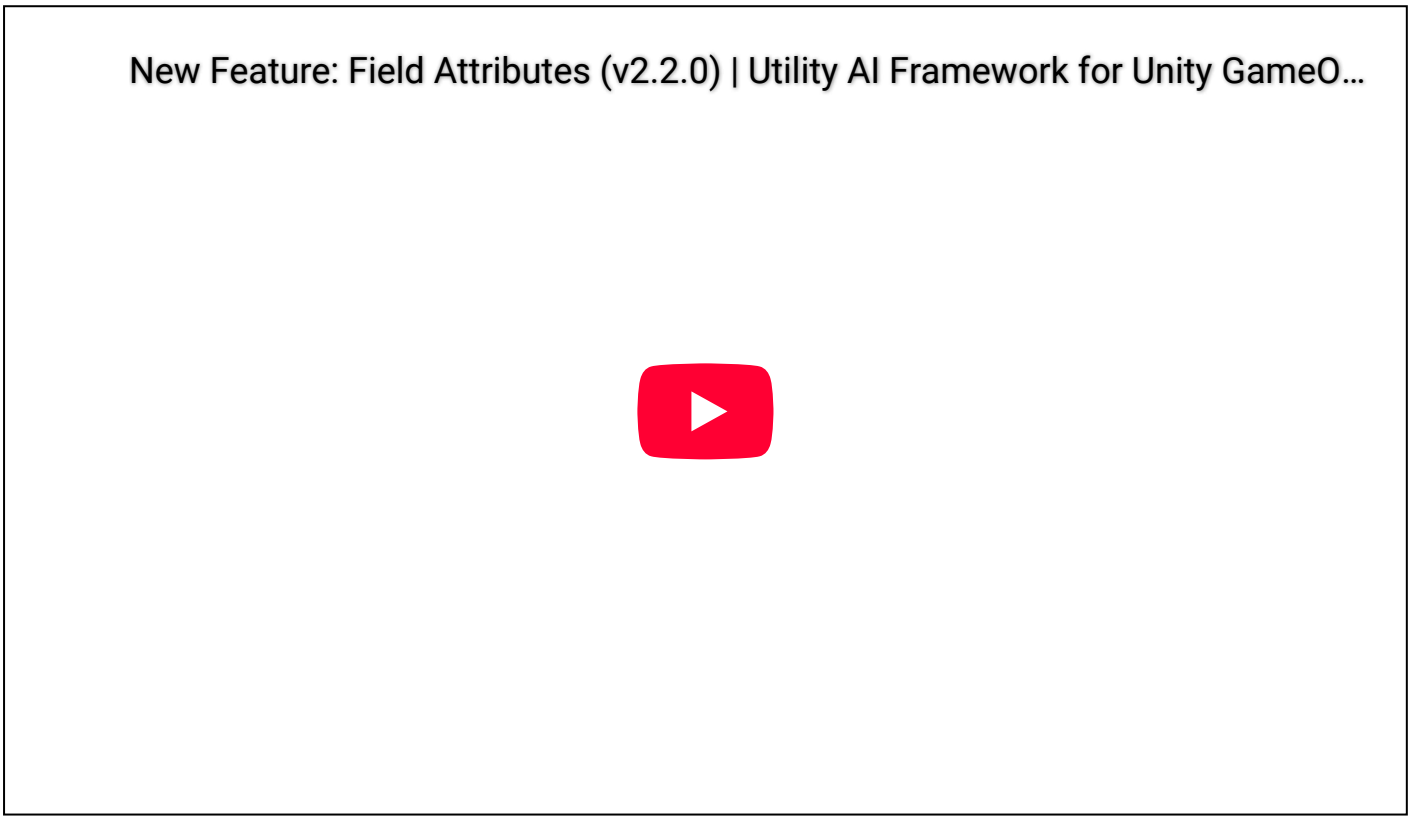
To change a field name from `OldField` to `NewField`, you need to pass the old field name to the constructor of `FieldFormerlySerializedAs`:

```
public class NewActionTask : ActionTask
{
    [FieldFormerlySerializedAs("OldField")]
    public int NewField;
}
```

Field Attributes

We have received feedback that it's currently hard to read our classes as they become more complex and have a lot of fields. So, in v2.2.0, we added these attributes to help you organize your fields in the **Intelligence Editor**.

> **Info**
>
> You can use these attributes in input, input normalizations, action tasks and target filters.

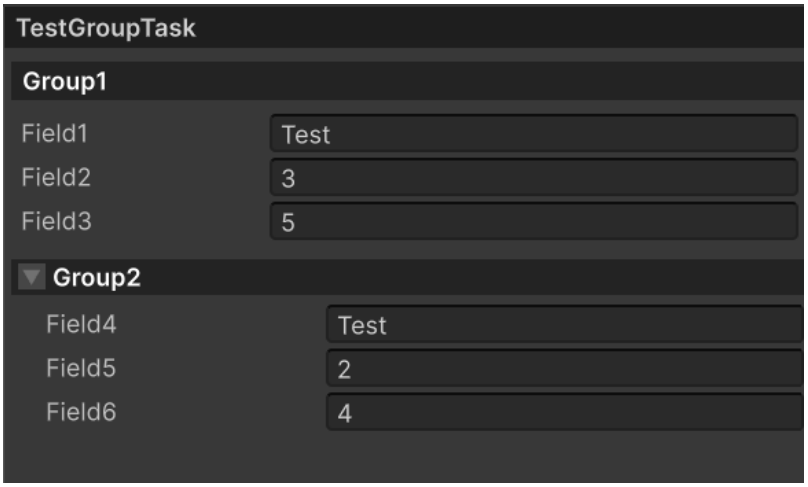


BoxGroup & FoldoutGroup

`BoxGroup` and `FoldoutGroup` attributes are used to group fields in the **Intelligence Editor**.

```
public class TestGroupTask : ActionTask
{
    [BoxGroup("Group1")]
    public string Field1;

    [BoxGroup("Group1")]
    public int Field2;

    [BoxGroup("Group1")]
    public float Field3;

    [FoldoutGroup("Group2")]
    public string Field4;

    [FoldoutGroup("Group2")]
    public int Field5;

    [FoldoutGroup("Group2")]
    public float Field6;
}
```

Here's how it looks in the **Intelligence Editor**:



ShowIf & HideIf

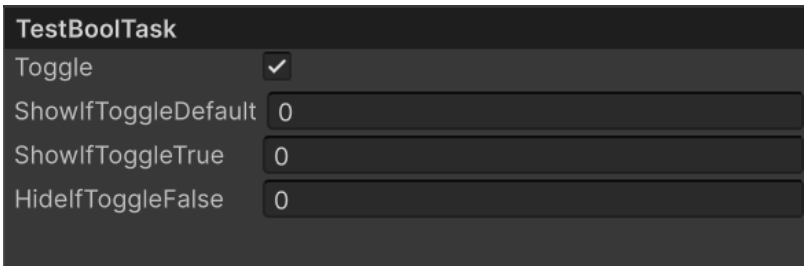
`ShowIf` and `HideIf` attributes are used to show/hide fields in the **Intelligence Editor**. These attributes allow users to display fields based on conditions. You can use them for basic types, such as `bool`, `enum`, `string`, `float`, and `int`.

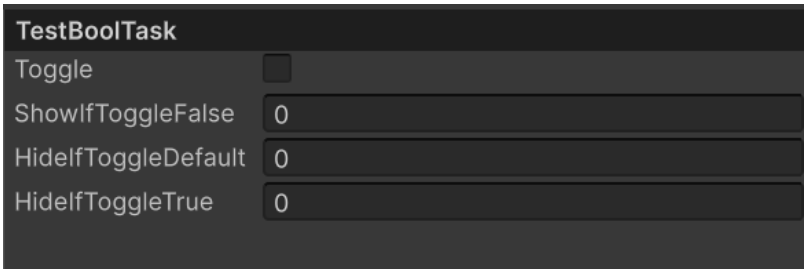
Here are examples of how to use these attributes with `bool` type and `enum` type:

Bool

```
public class TestBoolTask : ActionTask
{
    public bool Toggle;

    [ShowIf("Toggle")]
    public int ShowIfToggleDefault;

    [ShowIf("Toggle", true)]
    public float ShowIfToggleTrue;

    [ShowIf("Toggle", false)]
    public int ShowIfToggleFalse;

    [HideIf("Toggle")]
    public float HideIfToggleDefault;

    [HideIf("Toggle", true)]
    public float HideIfToggleTrue;

    [HideIf("Toggle", false)]
    public int HideIfToggleFalse;
}
```

Here's how it looks in the **Intelligence Editor**:

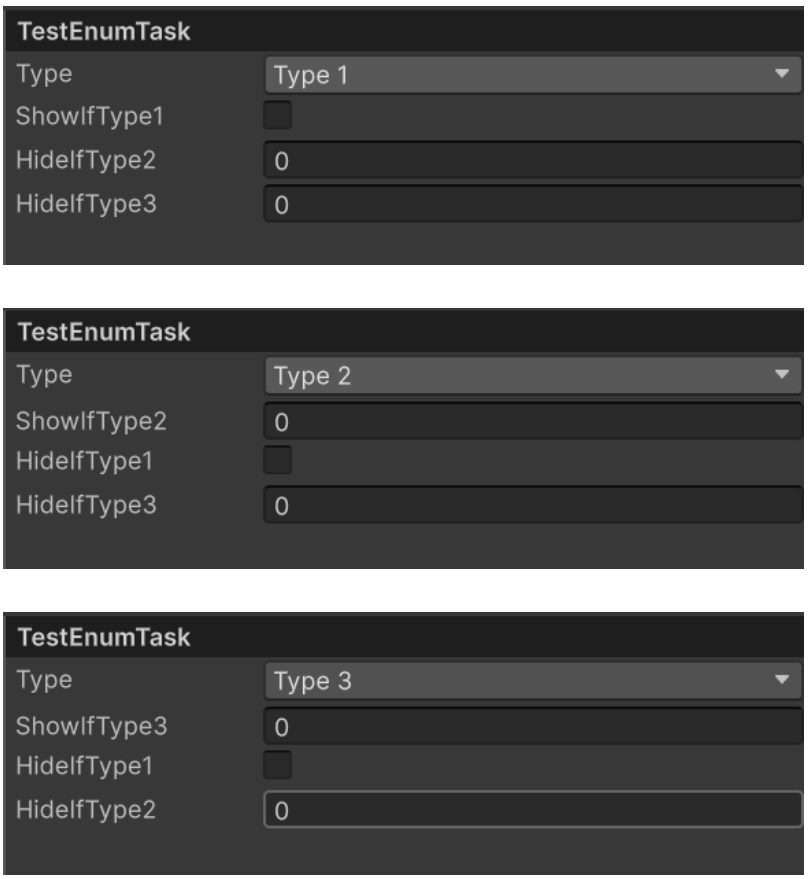




Enum

```
public enum TestEnum
{
    Type1,
    Type2,
    Type3,
}

public class TestEnumTask : ActionTask
{
    public TestEnum Type;

    [ShowIf("Type")]
    public bool ShowIfTypeDefault;

    [ShowIf("Type", TestEnum.Type1)]
    public bool ShowIfType1;

    [ShowIf("Type", TestEnum.Type2)]
    public float ShowIfType2;

    [ShowIf("Type", TestEnum.Type3)]
    public int ShowIfType3;

    [HideIf("Type")]
    public bool HideIfTypeDefault;
    [HideIf("Type", TestEnum.Type1)]
    public bool HideIfType1;

    [HideIf("Type", TestEnum.Type2)]
    public float HideIfType2;

    [HideIf("Type", TestEnum.Type3)]
    public int HideIfType3;
}
```

Here's how it looks in the **Intelligence Editor**:



Category Attribute

`Category` attribute is used to group your classes into categories. You can check how to use it here: Category Attribute.

Categories

As your AI system becomes more complex, you will have a lot of inputs, input normaizations, considerations and decisions, making it challenging to manage. That's why we provide these tools to help you group them into categories.

Category Attribute

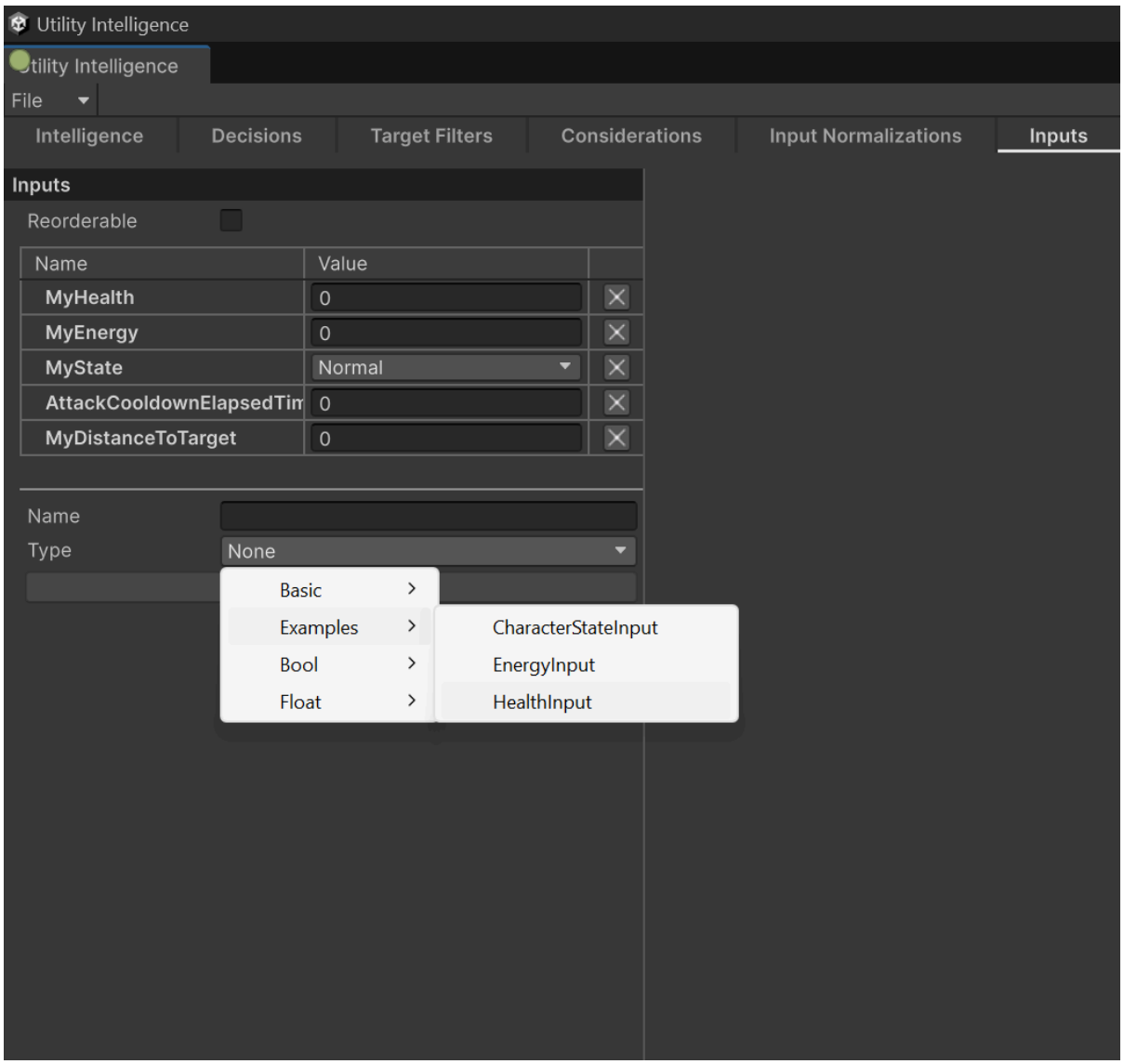
You can use the **Category Attribute** to group your classes into categories.

> **Info**
>
> The **Category Attribute** can be applied to input, input normalizations, action tasks, target filters and blackboard variables.

Here's an example of how to use it for inputs:

```
[Category("Examples")]
public class HealthInput : InputFromSource<int>
{
}
```

This allows you to group your inputs into categories in the **Input Type** dropdown menu within the **Input Tab**.

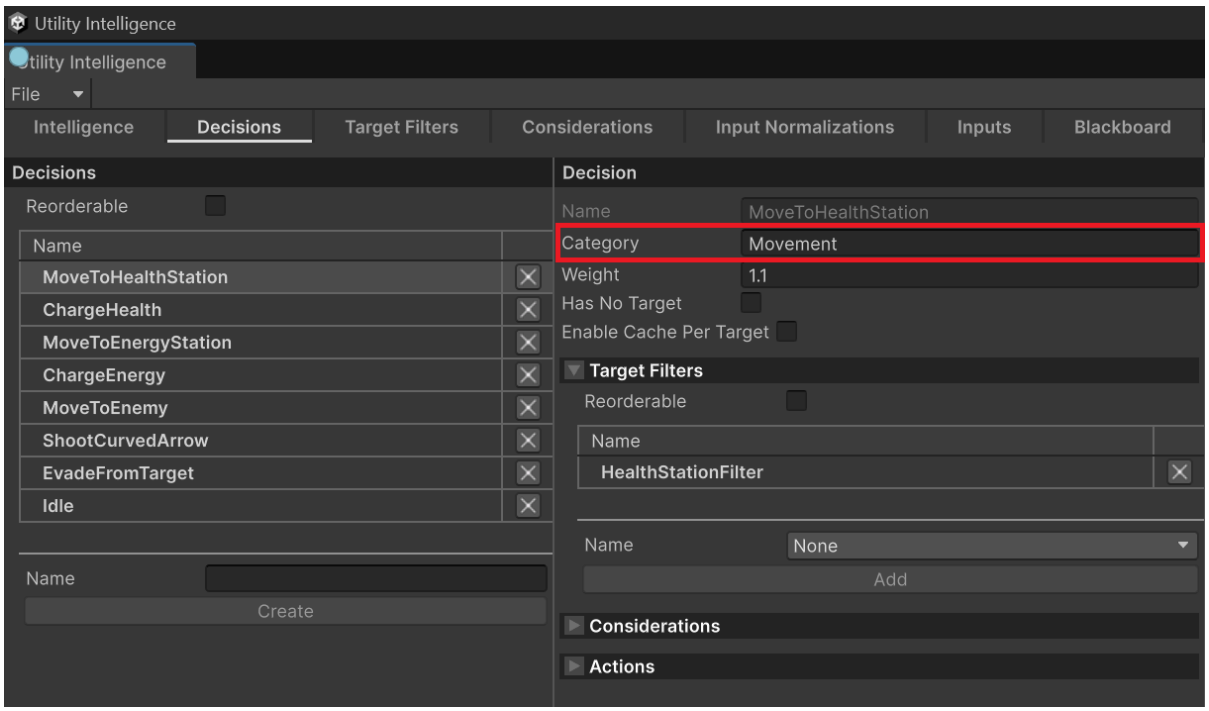


Category Field

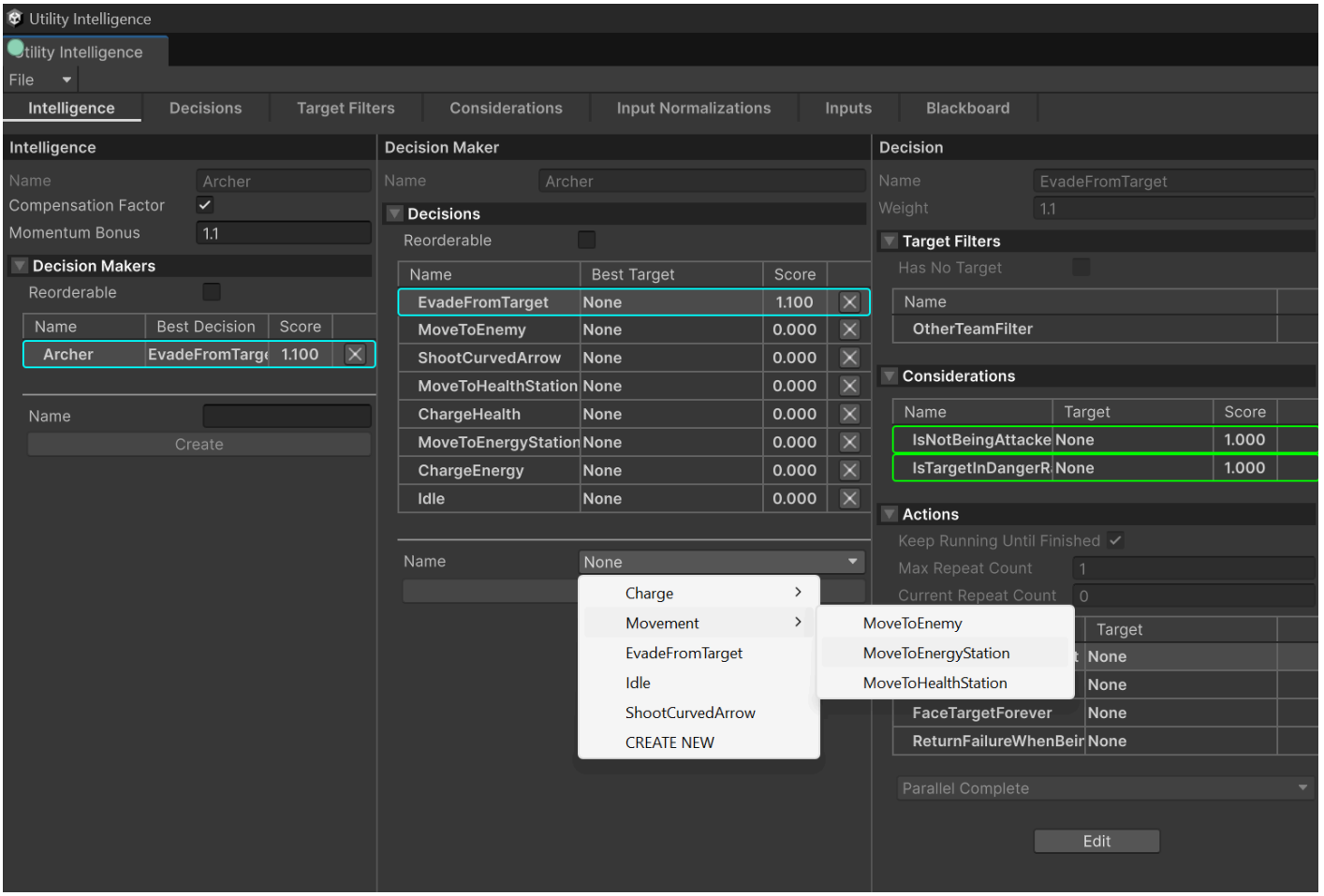
After they are created, you can use the **Category Field** to group them into categories.

> **Info**
>
> The **Category Field** is added in inputs, input normalizations, considerations, decisions, target filters and blackboard variables.

Here's an example of how to use it for decisions:



This allows you to group your decisions into categories in the **Decision Name** dropdown menu within the **Intelligence Tab**.



Upgrade Guide

General Upgrade Guide

Since the folder structure of this plugin might change frequently, the best way to upgrade **Utility Intelligence** to a newer version is to first delete your old asset folders and then re-import the new version:

1. Backup your project
2. Delete the following folders:
   - v1
     - Assets/CarlosLab/Common
     - Assets/CarlosLab/UtilityIntelligence
   - v2
     - Packages/com.carloslab.common
     - Packages/com.carloslab.utilityintelligence
3. Download the new version and then re-import the package.

However, if the changes are minor, such as upgrading from **v2.0.1** to **v2.0.2**, you can re-import the new version without having to delete the old asset folders.

Upgrading from v1 to v2

2.0.0 is a major release with a lot of changes. It includes some breaking changes that require manual updates when upgrading from v1 to v2. Sorry for the inconvenience.

> **Caution**
>
> These changes may break your project, so please backup your project before upgrading.

Intelligence Asset

We've made some breaking changes to the data structure of **Intelligence Assets** and increased the data version from v1 to v2. Therefore, you need to update your intelligence assets to data v2 so that this framework can deserialize them.

1. Update the intelligence data.
   - Select **File -> Show Data** to show the intelligence data.
   - Change `MyDistanceToTargetInput` to `DistanceToTargetInput`.
   - Change `NavMeshMoveTowards` to `MoveToTarget`.

- Remove all InputNormalizations from all Considerations.
- Move Decisions from DecisionMakers to the outer scope.
  Data Structure - v1

```
{
  "$type": "CarlosLab.UtilityIntelligence.UtilityIntelligenceModel",
  "DecisionMakers": [
    {
      "$type": "CarlosLab.UtilityIntelligence.DecisionMakerModel",
      "Id": "6f5616e5-a485-4c3b-9bc4-1eb1f10530fa",
      "Name": "Warrior",
      "Decisions": [
        {
          "$type": "CarlosLab.UtilityIntelligence.DecisionModel",
          "Id": "a36b4f16-d8d0-4069-94ab-925828eb3c7d",
          "Name": "MoveToHealthStation",
          ...
        }
      ],
      ...
    }
  ],
  ...
}
```

Data Structure - v2

```
{
  "$type": "CarlosLab.UtilityIntelligence.UtilityIntelligenceModel",
  "DecisionMakers": [
    {
      "$type": "CarlosLab.UtilityIntelligence.DecisionMakerModel",
      "Id": "6f5616e5-a485-4c3b-9bc4-1eb1f10530fa",
      "Name": "Warrior",
      ...
    }
  ],
  "Decisions": [
    {
      "$type": "CarlosLab.UtilityIntelligence.DecisionModel",
      "Id": "a36b4f16-d8d0-4069-94ab-925828eb3c7d",
      "Name": "MoveToHealthStation",
      ...
    }
  ],
  ...
}
```

- Select **File -> Import Data** to import the new intelligence data to the asset.

2. Create new input normalizations in the Input Normalization Tab.
3. Select the appropriate input normalization for your considerations in the Consideration Tab.
4. Add decisions to your decision makers in the Intelligence Tab.

Source Code

Input

Add the `in` keyword before `InputContext` in the `OnGetRawInput` function.

v1

```
protected override int OnGetRawInput(InputContext context)
```

v2

```
protected override int OnGetRawInput(in InputContext context)
```

InputNormalization

Change `InputContext` to `in InputNormalizationContext` in the `OnCalculateNormalizedInput` function.

v1

```
protected override float OnCalculateNormalizedInput(int rawInput, InputContext context)
```

v2

```
protected override float OnCalculateNormalizedInput(int rawInput, in InputNormalizationContext context)
```

Release Notes

Release Notes - v1

1.0.11

Changed

- Changed `GetVariable<TValue>()` function of the Blackboard to `GetVariable<TVariable>()`. Now, you have to pass variable type instead of value type to the function.

```
public void TestBlackboard()
{
    var blackboard = characterFacade.Entity.Intelligence.Blackboard;
    var sightRadiusVariable = blackboard.GetVariable<FloatVariable>("SightRadius");
    sightRadiusVariable.Value = 30;
}
```

Added

- Added `GameObjectListVariable` and `TransformListVariable` to store a list of GameObjects and Transforms in Blackboard.

Fixed

- Fixed a bug where the IntelligenceAsset did not save when changing Input to None.
- Fixed deserializing failed when a property value was null
- Fixed a bug where VariableReferences of Inputs had a null Blackboard at runtime.

1.0.10

Added

- Added `GetVariable<TValue>()` function for the Blackboard. You can use this function to retrieve Blackboard variables from other places.

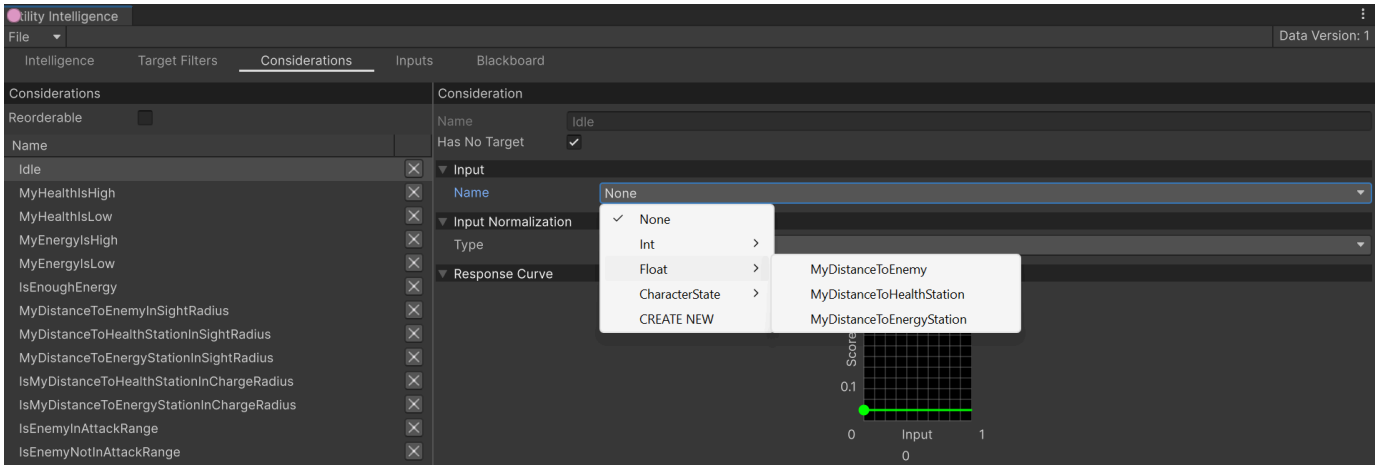
```
public void TestBlackboard()
{
    var blackboard = characterFacade.Entity.Intelligence.Blackboard;
    var sightRadiusVariable = blackboard.GetVariable<float>("SightRadius");
    sightRadiusVariable.Value = 30;
}
```

Fixed

- Fixed an issue that caused MomentumBonus to not work at runtime.

1.0.9

Changed

- In Unity 6, Unity has fixed the bug that prevented DropdownField choices from being nested. Therefore, we've updated our DropdownFields to include nested choices. If you use Unity 6, you will see some DropdownFields that have nested choices like this:
- 

1.0.8

Changed

- Refactored `Input` and `TargetFilter`.
- [Breaking] Renamed `IsLessThanOrEqualValueNormalization`s to `IsLessThanOrEqualToValueNormalization`s
- [Breaking] Renamed `IsGreaterThanOrEqualValueNormalization`s to `IsGreaterThanOrEqualToValueNormalization`s

> **Breaking Changes**
>
> Sorry, if you are using `IsGreaterThanOrEqualValueNormalization`s or `IsLessThanOrEqualValueNormalization`s, after upgrading to 1.0.8, you need to edit **Intelligence Data** to update these class names by using File Toolbar Menu.

Fixed

- Fixed an issue where the consideration editor did not update properly when removing an input from InputTab.

1.0.7

Fixed

- Fixed an issue where adding multiple target filters did not work

1.0.6

Added

- Added Ids for Views

Fixed

- Fixed an issue where list items could be renamed to an empty string.
- Fixed an issue where only the selected consideration would update the new input name when renaming an input.
- Fixed issues where only the selected decision would update the new consideration name when renaming a consideration, and the new target filter name when renaming a target filter.

1.0.5

Changed

- Group these classes under the menu: **AddComponent/CarlosLab**.
  - UtilityWorldController
  - UtilityAgentController
  - UtilityAgentFacade
  - UtilityEntityController
  - UtilityEntityFacade
- Separate the ChargeStations from Environment prefab in demos.

1.0.4

Added

- Added variable classes to store GameObject and Transform.

Fixed

- Fixed an issue where custom variables could not be referenced in the Editor.

1.0.3

Added

- Added Momentum Bonus to reduce the oscillation between nearly equal decision-target pairs.

1.0.2

Added

- A toggle to enable/disable Compensation Factor.

Changed

- Removed Consideration Benchmarks.

1.0.1

Added

- Consideration Benchmarks.
- InfluenceCurve Benchmarks.

Changed

- Select the first decision maker if all decision makers return a score of 0.

Fixed

- Fixed the issue where the state of a decision maker was incorrect when exiting/entering.
- Fixed the issue where the Editor did not select the correct decision when adding or removing decision makers, decisions and considerations.

1.0.0

First release

Release Notes - v2

2.2.6

Changed

- Updated **Documentation.pdf**

Fixed

- Fixed the issue where decisions without targets were run once per target. Now, they are only run once per decision-making update.

> **Caution**
>
> This update has changed some file names, so you must delete the old packages before upgrading. Check UpgradeGuide for instructions on how to upgrade.

2.2.4

Changed

- Unregistered `NormalizedInputChanged` in the `InputNormalizationItemViewModel` because it was unnesssary

Fixed

- Fixed a build error caused by including editor-specific code (`EnumFlagsField`) in the build process.

2.2.3

Changed

- Unbound cell items in ListViews.

Fixed

- Fixed issue where `asset = null` when exiting Play Mode and entering Edit Mode
- Fixed bug where the Consideration tab wasn't updating the response curve.

2.2.2

Changed

- Added support for Enums with the `FlagsAttribute` .

- Improved capturing of values from input fields (`IntegerField` , `FloatField` , `Vector3Field` , etc.) used for undo/redo functionality.

2.2.1

Added

- Added a new example to demonstrate the Entity Lifecycle. After upgrading, ensure to update the example scenes to access it.
- Added lifecycle event functions to `EntityFacade` . You can override these functions to receive notifications when lifecycle events occur.
  - `EntityFacade.OnRegistered()`
  - `EntityFacade.OnUnregistered()`
  - `EntityFacade.OnActivated()`
  - `EntityFacade.OnDeactivated()`
  - `EntityFacade.OnEnabled()`
  - `EntityFacade.OnDisabled()`
  - `EntityFacade.OnDestroyed()`
- Added support for `GameObject.SetActive` and `GameObject.Destroy` . Starting from v2.2.1, you can safely call these functions outside of action tasks. However, if you need to **activate/deactivate/destroy** utility entities within action tasks, you still have to use `EntityFacade.SetActive` and `EntityFacade.Destroy` . These functions will be queued to run after all action tasks have executed.
- Added functions that should be used from outside of action tasks. They will be run immediately without queueing.
  - `EntityController.RegisterImmediate()`
  - `EntityController.UnregisterImmediate()`
  - `EntityController.SetEnableImmediate()`
  - `EntityController.EnableImmediate()`
  - `EntityController.DisableImmediate()`
  - `EntityFacade.RegisterImmediate()`
  - `EntityFacade.UnregisterImmediate()`
  - `EntityFacade.SetEnableImmediate()`
  - `EntityFacade.EnableImmediate()`
  - `EntityFacade.DisableImmediate()`

Changed

- Group built-in Blackboard variables into categories by using the `Category` attribute.

Fixed

- Fixed a build error caused by including editor-specific code in the build process.

2.2.0

> **Breaking Changes**
>
> This version may break your project, so please back up your project before upgrading. Note that you should delete the old package folders first.

New Features

- Added a Lock Button to the **Intelligence Editor** to lock the editor window on a specific **Utility Agent**, preventing changes when clicking on another **Utility Agent** or **GameObjects**.
- Added JSON Attributes to rename fields, classes and namespaces in serialized **JSON**:
  - `ClassFormerlySerializedAs`
  - `FieldFormerlySerializedAs`
- Added Field Attributes to show/hide and group fields in the **Intelligence Editor**:
  - `ShowIf`
  - `HideIf`
  - `FoldoutGroup`
  - `BoxGroup`
- Added a Category Field to Inputs, Input Normalizations, Considerations, Target Filters, Decisions, Blackboard Variables to group them into categories in the **Intelligence Editor**.

Added

- Added the ability to close the **Intelligence Editor** by pressing the Escape button (Thanks David).
- Added the ability to rename list items (Decision Makers, Decisions, Considerations, …) by pressing the F2 button.
- Added support for `CategoryAttribute` in **Target Filters** and **Blackboard Variables**.

Changed

- Improved UI styles of **Inteligence Editor** (both **Dark** theme and **Light** theme).
- Moved `CategoryAttribute` from namespace `CarlosLab.UtilityIntelligence.Attributes` to namespace `CarlosLab.Common.Attributes`.

Fixed

- New Decision Makers, Decisions, Considerations, etc., cannot be created the first time after creating new Intelligence Assets.
- The **Runtime Editor** does not display the correct runtime theme.

2.1.1

Fixed

- Fixed a bug where the **Data Version Not Compatible** popup appears when clicking on a newly created Utility Intelligence Asset in Unity 6

2.1.0

New Features

- Add a new feature: **Decision Making Batch Processing**.

Added

- Added these new methods to safely activate/deactive utility entities.
  - `EntityController.SetActive(bool active)`
  - `EntityController.Activate()`
  - `EntityController.Deactivate()`
  - `EntityFacade.SetActive(bool active)`
  - `EntityFacade.Activate()`
  - `EntityFacade.Deactivate()`
- Added these new methods to safely enable/disable utility entities.
  - `EntityController.SetEnabled(bool enable)`
  - `EntityFacade.SetEnabled(bool enable)`
- Added these properties to retrieve information about utility entities:
  - `EntityController.IsRegistered`
  - `EntityController.IsActive`
  - `EntityController.IsEnabled`
  - `EntityController.IsDestroyed`
  - `EntityFacade.Id`
  - `EntityFacade.IsRegistered`
  - `EntityFacade.IsEnabled`
  - `EntityFacade.IsDestroyed`

- Added `EntityFacade.DestroyAfter()` to destroy entities with a delay.

Changed

- Updated **UtilityAgentSpawner** example : Increased the map size to spawn hundreds of agents for testing the decision-making batch processing.
- Updated `Documentation.pdf` to the newest version.
- Removed the ability to enable/disable utility entities through `GameObject.SetActive()` because it is not safe when called from action tasks. Instead, use `EntityController.SetActive()` or `EntityFacade.SetActive()`.
- Set the execution order of world controllers to `-100` to make it run before all other scripts.
- Restricted names of target filters, decision makers, decisions, considerations, inputs, and input normalizations to allow only letters, numbers, underscores and and a maximum length of **64** characters.
- Renamed `World.ActiveEntities` to `World.EnabledEntities`
- Renamed `UtilityWorld.ActiveAgents` to `UtilityWorld.EnabledAgents`
- Exposed `UtilityWorld.EnabledAgents` as a **public** property
- Changed the text of serialized generic types:
  - **2.0.x**:

    ```
    CarlosLab.Common.VariableReference`1[[System.Int32]]
    ```

  - **2.1.0**:

    ```
    CarlosLab.Common.VariableReference`1[System.Int32]
    ```

Fixed

- Fixed a bug where enabling/disabling utility entities from action tasks could break the decision-making process by throwing **InvalidOperationException: Collection was modified**;
- Fixed a bug where disabling an agent did not abort its current decision, causing it to continue running the decision's action tasks.
- Fixed a bug where `VariableReference` with an array value type could not be serialized.

Removed

- Remove the **FrameworkVersion** from **UtilityIntelligenceModels** because it was unnecessary.

> ⚠ **Backup your project before upgrading!**
>
> Please backup your project before upgrading. This version changes how generic types are serialized. Although it is automatic, it might still cause unexpected issues for unforeseen reasons.

> **Note**
>
> For those using `GameObject.SetActive()` to activate/deactivate utility entities, you have to switch to using `EntityController.SetActive()` or `EntityFacade.SetActive()` instead to safely activate/deactivate utility entities.

2.0.4

Added

- Added **ScriptableObjectVariable** and **ScriptableObjectListVariable** to store ScriptableObjects in Blackboard.

Improved

- Improved TargetFilters' performance.

Changed

- Disabled clearing of the **Utility Intelligence Editor** when selecting a non-agent GameObject.

Fixed

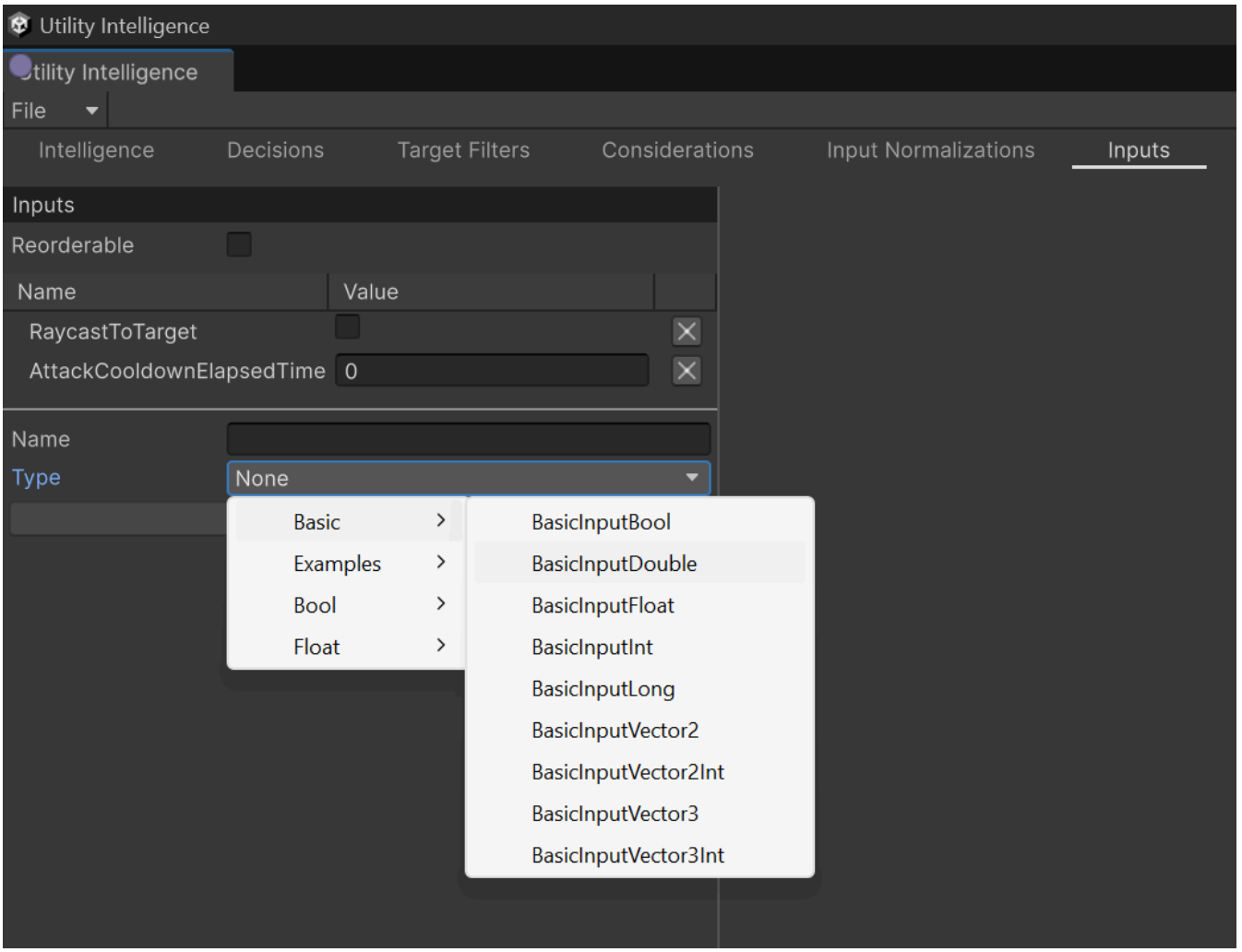
- Fixed a bug where the current decision does not break its current action to switch to the best decision when the "Keep Running Until Finished" option is not ticked.

2.0.3

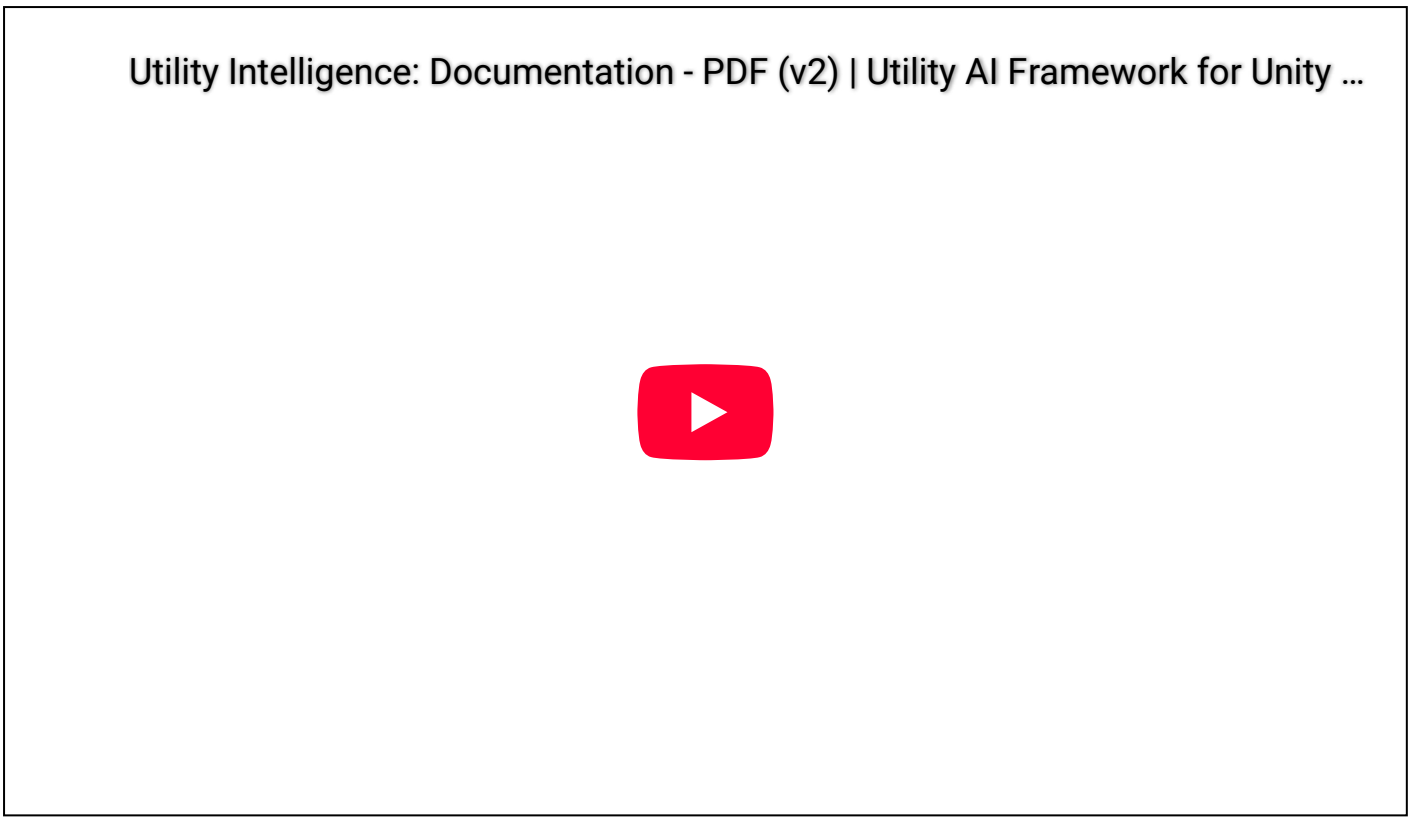
Added

- Added a bunch of new basic inputs that retrieve values from Blackboard:
  - BasicInputInt
  - BasicInputBool
  - BasicInputFloat
  - BasicInputDouble
  - BasicInputLong
  - BasicInputVector2
  - BasicInputVector3
  - BasicInputVector2Int

- BasicInputVector3Int



- Added a PDF version of the documentation, so you can now read it offline without needing an internet connection.



Fixed

- Fixed a bug where the framework could not deserialize inputs if their values types were changed, such as from `Input<bool>` to `Input<float>`.

2.0.2

Fixed

- Fixed bug where the **File Menu Toolbar** could not be used because the **Data Version is Not Compatible** popup showed repeatedly if the data version of **Utility Intelligence Assets** was older than the framework.
- Fixed **NullReferenceException** that occurred when agents made decisions at runtime if the decision list of decision makers was empty.
- Fixed **NullReferenceException** that occured when using the **File Menu Toolbar** without selecting a **Utility Intelligence Asset**.

Removed

- Removed the **FrameworkVersion** from **Utility Intelligence Assets** because it was unnecessary.

2.0.1

Fixed

- Fixed a bug where removing a consideration in the **ConsiderationTab** did not clear the **ConsiderationView** in the **IntelligenceTab** and caused a `NullReferenceException` in `InputNormalizationViewIntelligenceTab` and `InputValueViewIntelligenceTab` when trying to access the removed consideration view model.
- Fixed a bug where removing in the **DecisionTab** did not clear the **ActionTaskView** in the **IntelligenceTab**

2.0.0

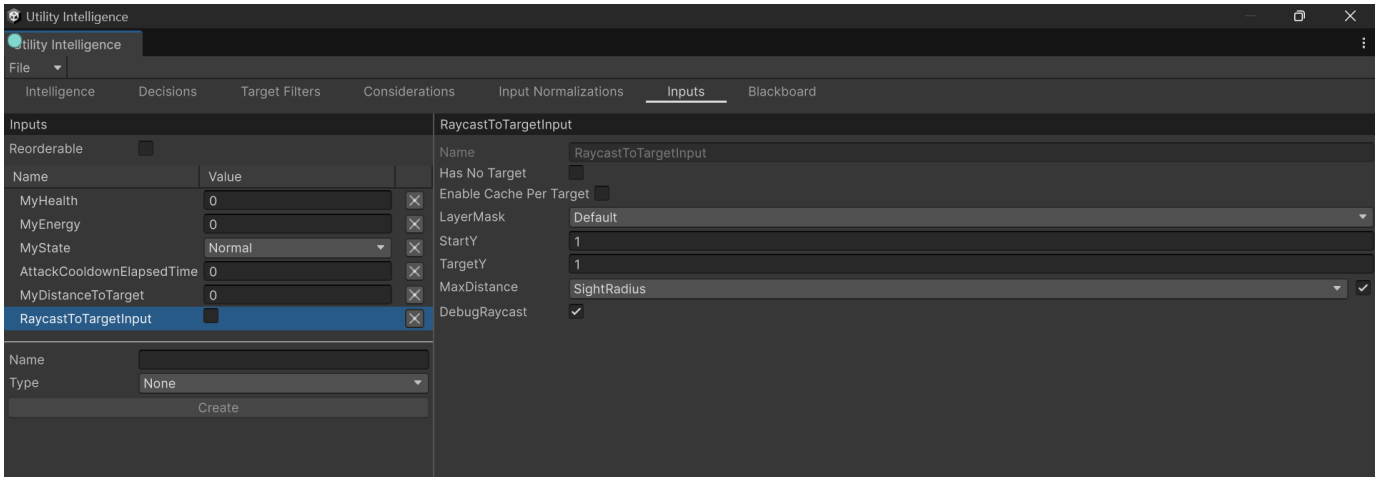
> **Note**
>
> Starting with v2, this plugin has been moved from the **Assets** folder to the **Packages** folder to manage dependencies and track versions more easily.

To upgrade from v1 to v2, please read the Upgrade Guide.

New Features

- Add a new feature: Runtime Editor

Added

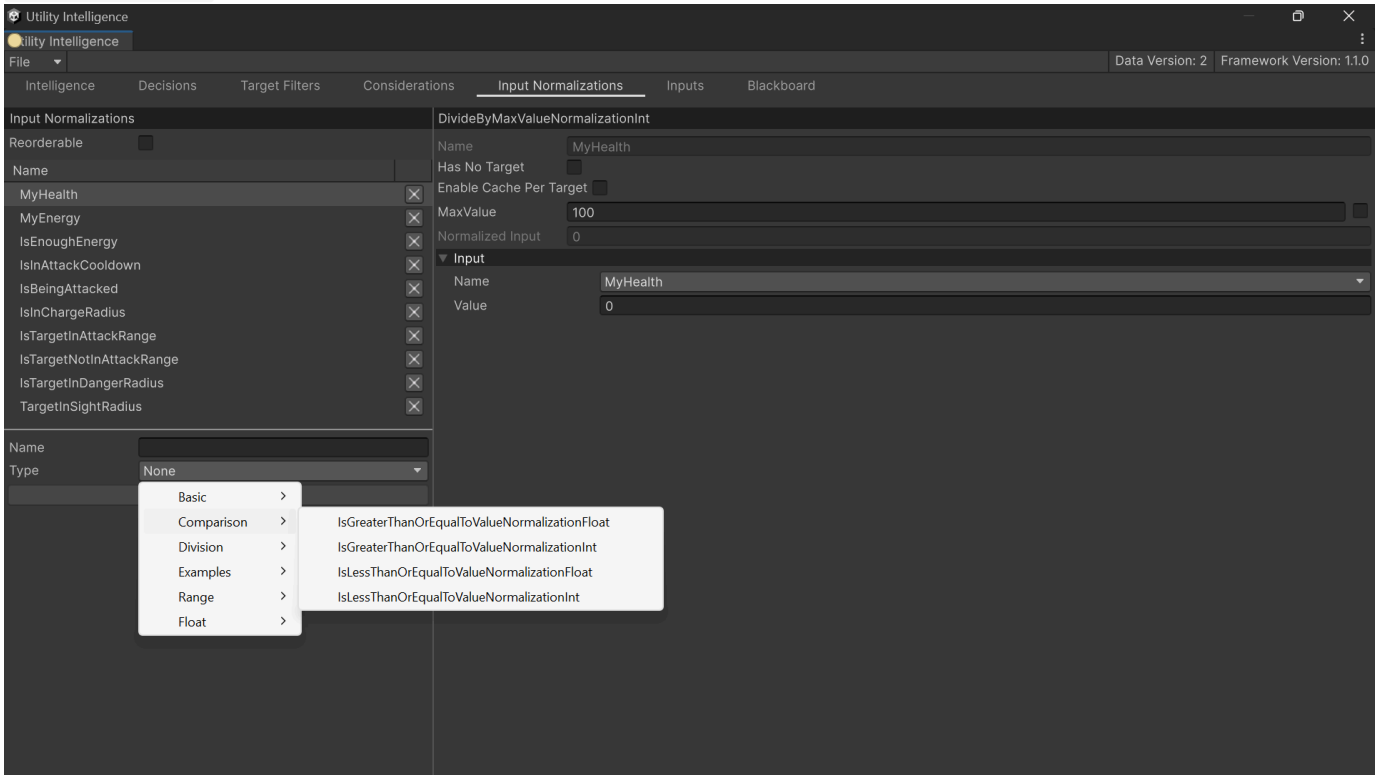
- Added a lot of new ExampleScenes.
- Added two new tabs to the Intelligence Editor: Decision Tab and Input Normalization Tab
- Added **HasNoTarget** and **EnableCachePerTarget** toggles for decisions, considerations, input normalizations and inputs to enable caching of their calculated results, thereby eliminating unnecessary recalculations.
- Supported serializing **LayerMask**. Starting from v2, you can edit all `LayerMask` fields in the **Utility Intelligence Editor**, and all the changes will be serialized to **JSON** and saved to Utility Intelligene Asset.
- 

- Added `NavMeshAgentVariable` and `AnimatorVariable` to store `NavMeshAgent` and `Animator` in Blackboard
- Added `CategoryAttribute` to categorize the action tasks, inputs and input normalizations.
- Added `Blackboard.TryGetVariable()`
- Added these new methods to safely Enable/Disable utility entities.

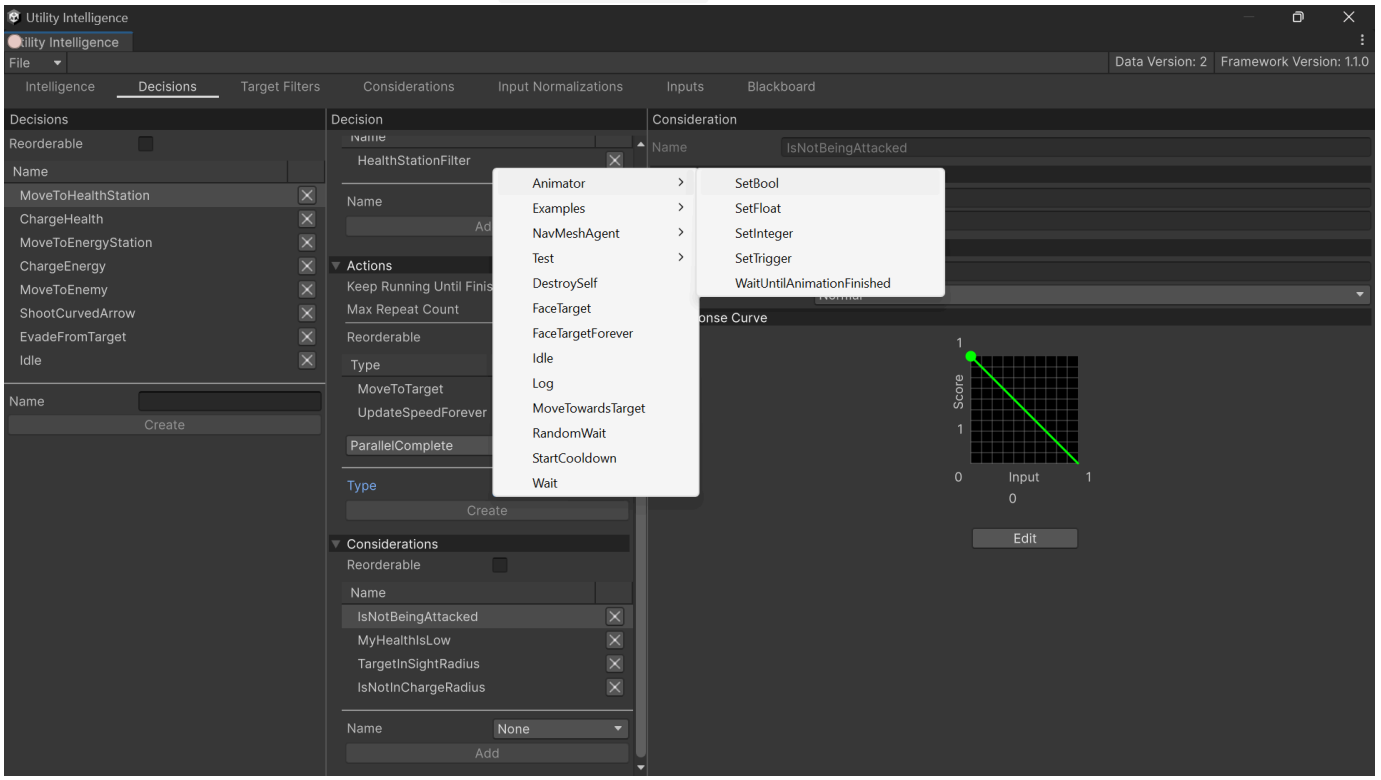
  - `EntityController.Enable()`
  - `EntityController.Disable()`
  - `EntityFacade.Enable()`
  - `EntityFacade.Disable()`
- Add some new target properties to the action tasks:
  - TargetFacade
  - TargetAgent
  - TargetEntity
  - TargetTransform
  - TargetGameObject
- Added a lot of new classes to the built-in library:
  - Action Tasks
    - Animator
      - SetBool
      - SetFloat
      - SetInteger
      - SetTrigger
      - WaitUntilAnimationFinished
    - NavMeshAgent
      - ChaseTarget
      - MoveAwayFromTarget
      - Patrol
    - FaceTarget
    - FaceTargetForever
    - StartCooldown
  - Inputs
    - CooldownElapsedTimeInput
    - RaycastToTargetInput
  - Input Normalizations
    - IsInCooldownNormalization
- Added **DecisionInfo** prefab to show which decision has been chosen.

Improved

- Improved performance when calling Unity Event Functions in action tasks. Previously, they were called for all the action tasks across all decisions. Currently, we only call them for the action tasks of the chosen decision.
  - LateUpdate
  - FixedUpdate
  - OnCollisionEnter
  - OnCollisionStay
  - OnCollisionExit
  - ...
- Categorize the inputs, input normalizations based on its input value type and `CategoryAttribute`. Note that the `CategoryAttribute` will take priority.



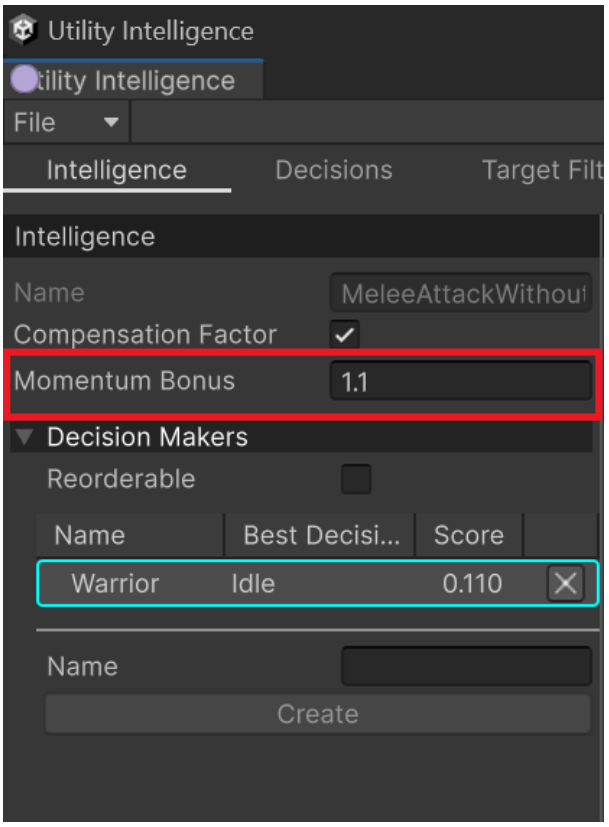
- Categorize the action tasks based on its `CategoryAttribute`.



- Auto save the widths of the panes in the Utility Intelligene Editor after they have been resized.

Changed

- The Momentum Bonus is no longer fixed at **25%**. Now, you can adjust it as desired.



- Renamed

  - `NavMeshMoveTowards` to `MoveToTarget`
  - `MyDistanceToTargetInput` to `DistanceToTargetInput`
  - `MoveTowards` to `MoveTowardsTarget`
- Serialization: When changing type of properties in Inputs, InputNormalizations, ActionTasks and TargetFilters, their values will be reset to the default value of their types.
- Disable Runtime Editing of Decisions, Considerations for safety purposes, as they are prone to errors.
- Disable preview of consideration info when it is discarded in Editor mode.

Fixed

- Fixed a bug where the UtilityIntelligenceEditor did not clear the view when exiting Runtime Mode
- Fixed a bug where renaming a consideration in the Consideration Tab did not update the new consideration name in Decision Tab and Intelligence Tab.
- Fixed a bug where renaming a target filter in the Target Filter Tab did not update the new target filter name in Decision Tab and Intelligence Tab.
- Fixed a bug where renaming and removing a Blackboard variable in the Blackboard Tab did not update the variable references in inputs, input normalizations, target filters and decision tasks.
- Fixed a bug where we cannot undo or redo the `ActionExecuteMode` after it has been changed.
- Fixed the delay when transitioning between decisions and action tasks when they are running in sequence.

FAQs

FAQs

Why use Utility Intelligence

I created a page to explain the benefits of using **Utility Intelligence** over other tools on the market. You can check it out here: Why Use Utility Intelligence?

Which Unity version is supported?

Utility Intelligence is designed for **Unity 6 and later**. It uses the **UIToolkit Runtime Binding System** introduced in **Unity 6** to build the **Intelligence Editor** using the **MVVM pattern**, which allows it to **function** not only **in the Editor** but also **at runtime in builds**. Therefore, **Unity 6 and higher** is **required**.

How to get support?

Currently, there are 3 ways to get support:

1. Official Support
   - Join my community and post your questions there: **Join Us On Discord**.
     - This requires you to **verify** your **InvoiceNumber (OrderId)** first to gain **access** to the **private channels**.
     - This is the recommended way to get support because:
       - You can find instant answers to your questions by searching through old posts if someone has already asked the same questions as you.
       - Your questions also help others, as they won't need to ask the same questions again.
       - You will get faster answers than by sending support requests via email.
   - Send an email to support@carloslab-ai.com, and don't forget to include your **Invoice Number (OrderId)**.
     - Only use this if you really don't like using **Discord**.
2. Community Support
   - If you don't want to wait for offcial support, you can ask for support from the community. You can post your questions on:
     - Unity Discussions: Utility Intelligence: A Robust And Powerful Utility AI Framework.
     - The **community-support** channel in our **Discord** server.

Why you should join our community on Discord

- We have a dedicated channel for sharing knowledge about using Utility Intelligence (**knowledge-base**), where you can both share and learn from others in our community.
  - Additionally, I will add all valuable content in the channel to our documentation. This will give your AI Assistant a more extensive knowledge base about Utility Intelligence, making it smarter. This benefits everyone.
- We have a dedicated channel for receiving feedback from users, where you can send feedback to us. If it is reasonable and within our capabilities, we'll make improvements to our framework based on your suggestions.
- You can ask for support either from us (official) or from the community.
- You can find instant answers to your questions by searching through old posts, without having to wait for support.
- You will get faster responses than by asking via email.

Why use Utility Intelligence?

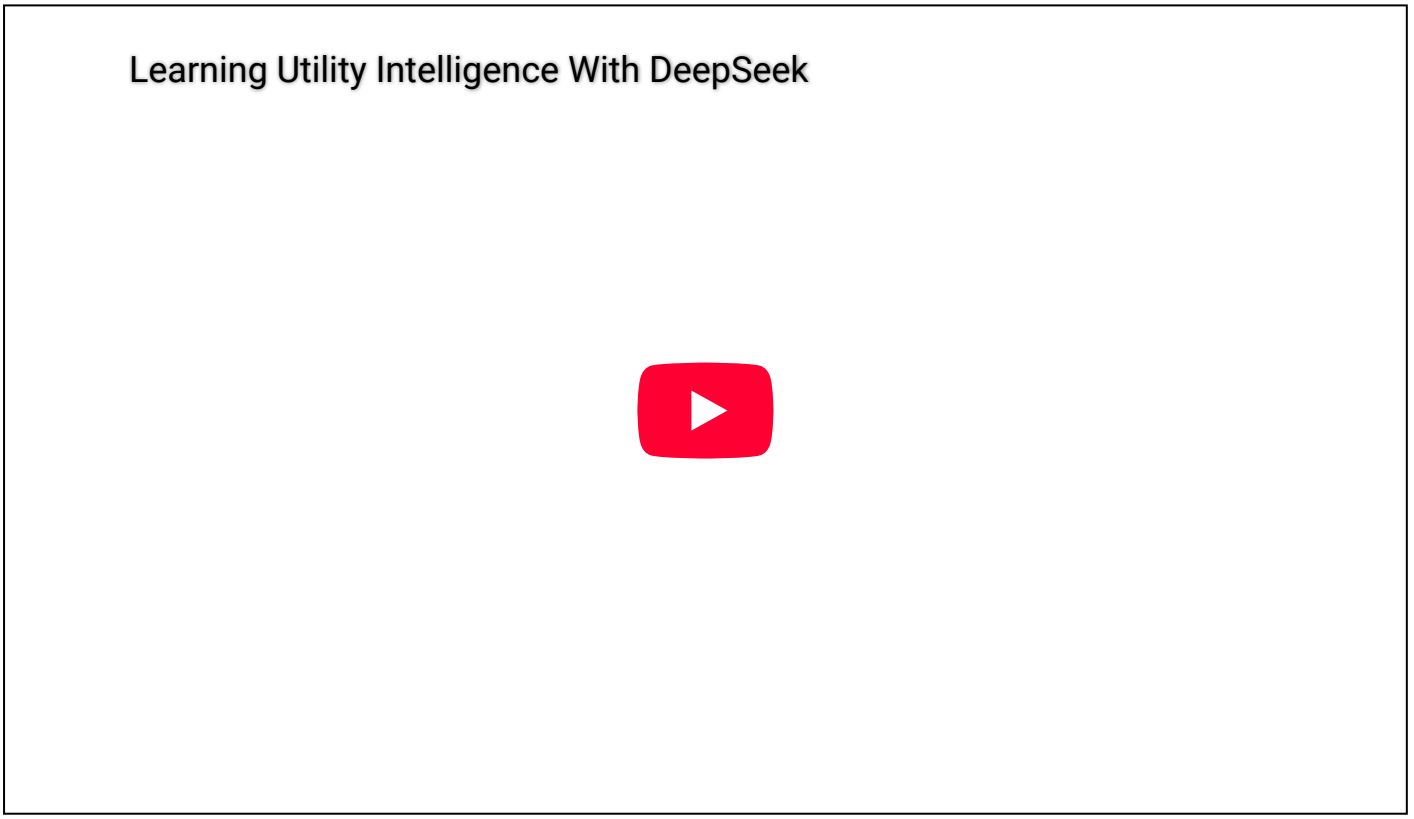
High-quaility documentation

The **documentation** is written with **care** and is **regularly improved** to help you learn **Utility Intelligence** as **easily** as possible.
Besides the **online** version, we also have a **PDF** version for **offline** reading. You can feed it to any **AI chatbot**, and then ask it any **questions** you have about **Utility Intelligence**.
-> **No more struggling** with **low-quality** documentation that makes you **feel frustrated** and **wastes** your **time**.

- **Online**: https://utilityintelligence.carloslab-ai.com/Documentation/
- **Offline**: https://utilityintelligence.carloslab-ai.com/assets/Documentation/Documentation.pdf

I created this video to show you how to learn **Utility Intelligence** with **DeepSeek**. I like it because its deep thinking mode is incredible. If you don't like DeepSeek, you can use any other AI chatbot you prefer.

But don't forget to share your knowledge about **Utility Intelligence** in the channel: **knowledge-base** on our Discord server. I will select the most valuable content to include in our documentation. This will give your AI Assistant a larger knowledge base and be smarter.



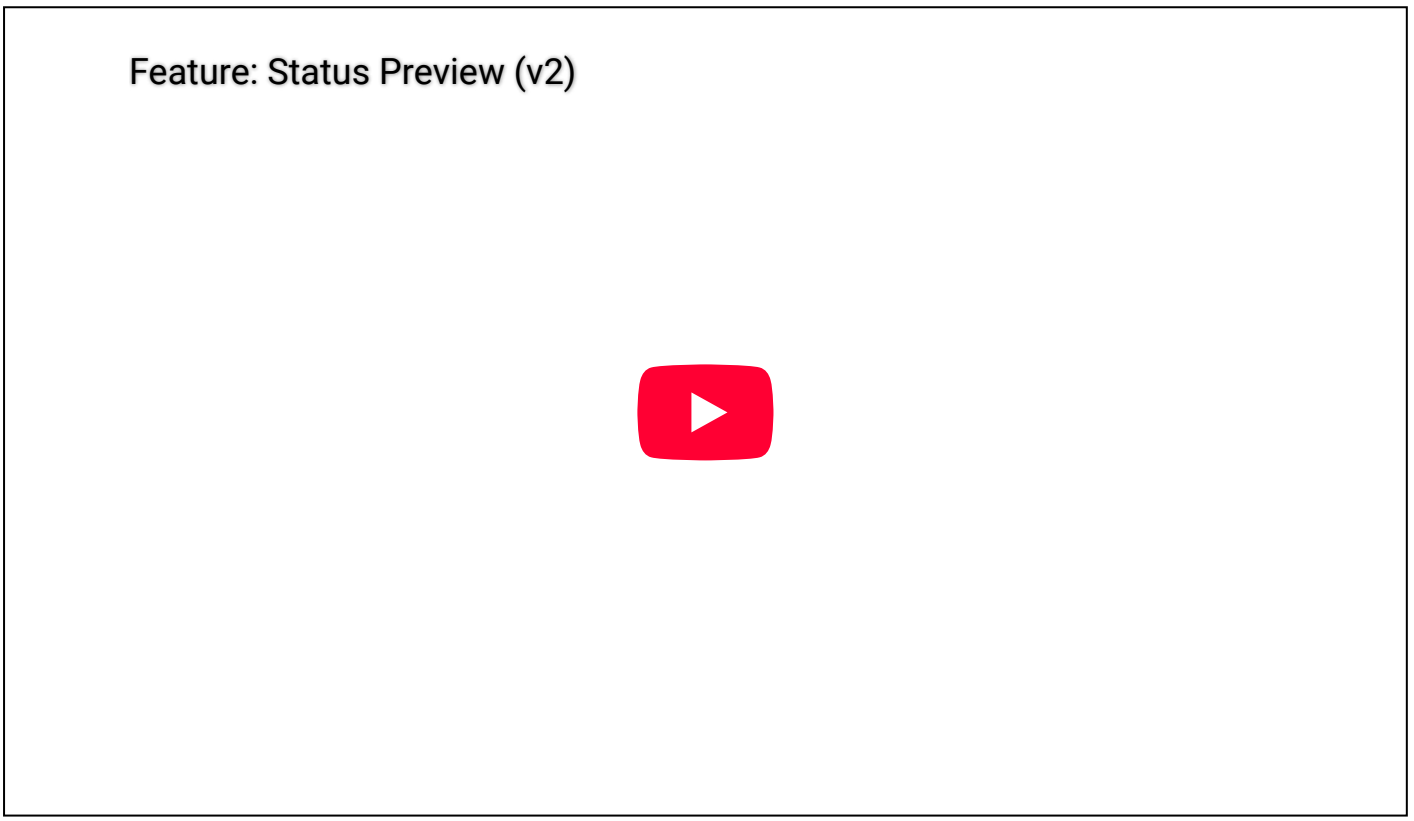
Utility AI is better than Behavior Trees and Finite State Machines

I've written an article to explain why **Utility AI** is better than **Behavior Trees** and **Finite State Machines** for creating game AIs, you can read it here: Why use Utility AI instead of Behavior Trees and Finite State Machines to create AIs for your games.

Easy to debug

If you use **Behavior Trees** or **Finite State Machines** as your **decision-making** solution, you might find it hard to debug why your agents make wrong decisions at runtime as complexity increases.

With **Utility Intelligence**, you can **preview** which **decision** is **chosen** by modifying **input values,** such as health, energy, distance to target, and attack cooldown, directly **in the Editor**, **without** having to **play** the game.



Easy to maintain and scale

if you use **Behavior Trees** or **Finite State Machines** for **decision-making**, the cost of maintaining the behavioral structure will increase as the complexity of AI Behaviors increases. It is because the temporal coupling between decisions.

In **Utility Intelligence**, we use **Utility AI** for **decision-making**, which means **decisions** are **made** based on their **scores**. Therefore, there is **no coupling** between **decisions**, and they are **independent** of each other.

-> It's easy to **add**, **remove** and **change** decisions, as well as **adjust decision-making** by **tweaking** the decision **scores**, **without worrying** about causing **significant changes** to the **behavioral structure**, as in **Behavior Trees** and **Finite State Machines**.

-> This ensures that your **AI system** remains **manageable** and **scalable** as its **complexity increases**.

Boost team productivity

Since decisions are made based on their scores, **designers** can **adjust decision-making** by **tweaking** the decision **scores**, **without** needing **support** from **developers** to **change** the **behavioral structure**, as required in **Behavior Trees** and **Finite State Machines**.
-> **Designers** and **developers** can **work independently** without affecting each other.

- **Designers**: Focus on adjusting the decision scores to ensure the best decision is chosen in any situation.
- **Developers**: Focus on creating and executing new decisions based on the game design document.

Higher Performance

Utility AI allows us separate **decision-making** from **decision-execution**, turn them into two distinct processes, and run each process at a **different frequency**.

For example, we can run the **decision-execution** process **every frame** while running the **decision-making** process only **every 0.1s** or **every 0.5s** by adjusting the decision-making interval to suit your game's needs.

Moreover, you can even distribute the decision-making process across multiple frames to balance the workload, or **manually** run the **decision-making** process when necessary. This approach significantly improves your game's performance.

This is difficult to achieve if you use **Behavior Trees (Finite State Machines)** because decision-making is closely tied to decision-execution by nature in these systems and it's hard to separate.

An intuitive and powerful Editor

We offer an **intuitive** and **powerful Editor** with many **robust features** that allow you to create **complex** AI Behaviors and Logic **with ease**:

- **Status Preview**: Preview the score of each decision and which decision is chosen based on the input values and response curves directly in the Editor, without having to play the game.
- **Consideration Editor**: See how the input and the response curve will affect the consideration score without having to visualize it in your head.
- **JSON Editing**: Manually edit the **Intelligence Data** in JSON format using your Text Editor then import it to the **Intelligence Asset**
- **Runtime Status**: View the current status of multiple components during runtime. It is similar to **Status Preview** but includes additional runtime information, such as the best target for each decision, and the current status of considerations and action tasks.
- **Runtime Editing**: Tweak your AI Behaviors during runtime for testing purposes without having to replay the game.
- **Runtime Editor**: The **Utility Intelligence Editor** can function both **at editor time** and **at runtime in builds**. This feature enables users to adjust variables in the **Utility Intelligence Editor** to observe how they affect the

agent's decisions for testing purposes in builds.

- **Lockable Editor**: Lock the **Intelligence Editor** on a specific **Utility Agent**, allowing users to modify variables from other **Game Objects** through the **Inspector Window** and see how they affect the decision scores in the **Intelligence Editor**.
- **Field Attributes**: Show/hide and group your fields in the **Intelligence Editor**.
- **Dark** & **Light themes**: The **Utility Intelligence Editor** supports both Dark and Light themes and will automatically match the theme of the Unity Editor.

Many example scenes

We offer many **example scenes** to show you how to use **Utility Intelligence** to create AIs for your games:

- StraightArrowOnly
- StraightArrow vs CurvedArrow
- Chaser vs Evader
- Chaser & Patrol vs Evader & FindEnemy
- Swordsman vs Swordsman
- Axeman vs Axeman
- Archer vs Swordsman
- Crossbowman vs Swordsman
- Team vs Team
- Runtime Editor

Many built-in components

We offer many **built-in** components to help you create game AIs more **easily** and **quickly**, saving you a **significant** amount of **time**:

- Built-in Inputs
- Built-in Input Normalizations
- Built-in Action Tasks
- Built-in Target Filters
- Built-in Blackboard Variables

Many optimization tricks

We offer many **optimization tricks** to help you **discard unnecessary calculations** and **improve** your AI's **performance**:

- **Discard** considerations if the decision cannot possibly beat the other ones.
- **Discard** decisions, decision makers if they cannot possibly beat the other ones.
- **Cache** the calculated results from inputs, input normalization, considerations, decisions and reuse them in other places.
- **Adjust** the decision-making interval, such as **0.1s** or **0.5s**, depending on your game's needs.
- **Distribute** the decision-making task across multiple frames to balance the workload, reduce computational burden per frame, and avoid performance spikes.

Many oscillation reduction tricks

We offer many **oscillation reduction tricks** to minimize the oscillation between decisions:

- Momentum **Bonus**
- Decision **Weight**
- **Keep running** the decision tasks until they are **finished**

Special Thanks

This framework is inspired by these projects:

1. **Infinite Axis Utility System** (*Dave Mark*). For more information about it, you can watch his presentations here.
2. **Curvature** (*Mike Lewis*)

Special thanks to **Dave Mark**, and **Mike Lewis** for their inspiring work.

Third Party Notices

Framework

This framework uses some components from the following projects:

1. Curvature (Mike Lewis)
   - Component: ResponseCurve.cs
   - Url: https://github.com/apoch/curvature/
   - License: BSD-3
2. Trove (PhilSA)
   - Component: CurveDrawerElement.cs
   - Url: https://github.com/PhilSA/Trove/
   - License: MIT

Many thanks to **Mike Lewis**, and **PhilSA** for creating these excellent tools.

Example Scenes

This package uses the following assets to create example scenes:

1. KayKit - Character Pack : Adventurers (Kay Lousberg)
   - Url: https://kaylousberg.itch.io/kaykit-adventurers
   - License Type: CCO
2. KayKit - Dungeon Remastered Pack (Kay Lousberg)
   - Url: https://kaylousberg.itch.io/kaykit-dungeon-remastered
   - License Type: CCO
3. KayKit - Character Pack : Skeletons (Kay Lousberg)
   - Url: https://kaylousberg.itch.io/kaykit-skeletons
   - License Type: CCO
4. KayKit - Mini Game Variety Pack (Kay Lousberg)
   - Url: https://kaylousberg.itch.io/kay-kit-mini-game-variety-pack

- License Type: CCO

5. RPG Audio (Kenney)
   - Url: https://kenney.nl/assets/rpg-audio
   - License Type: CCO
6. Impact Sounds (Kenney)
   - Url: https://kenney.nl/assets/impact-sounds
   - License Type: CCO
7. 3D Game Kit (Unity)
   - Component: Audios
   - Url: https://assetstore.unity.com/packages/templates/tutorials/unity-learn-3d-game-kit-115747
   - License Type: Unity Companion License
8. Dragon Crashers (Unity)
   - Component: Audios
   - Url: https://assetstore.unity.com/packages/essentials/tutorial-projects/dragon-crashers-urp-2d-sample-project-190721
   - License Type: Unity Companion License

Many thanks to **Kay Lousberg**, **Kenny**, and the **Unity Asset Team** for creating these excellent assets.