Malgam Documentation (by NikosAssets)

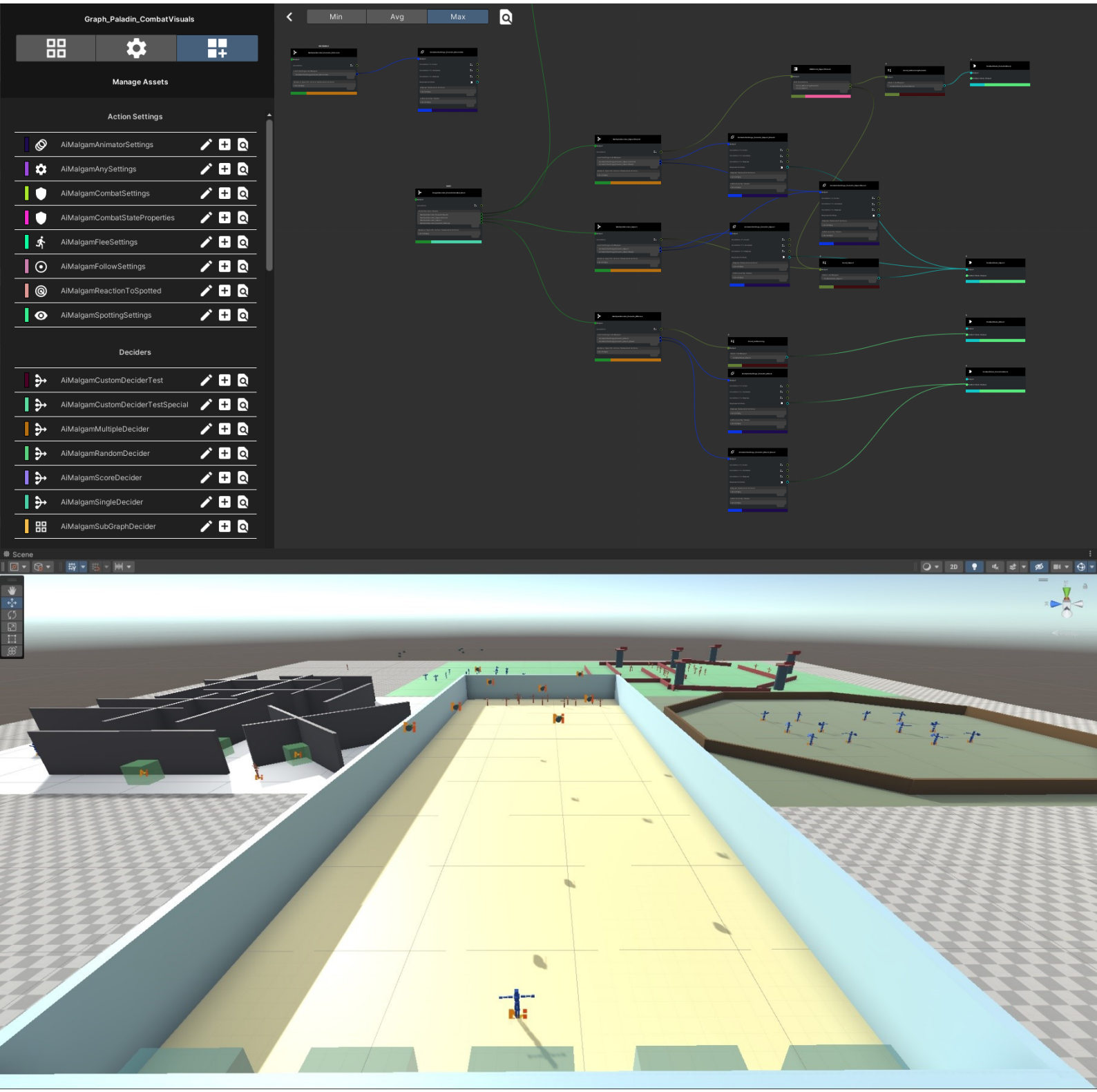


Table of Contents

1. About AiMalgam

This asset pack offers a **generic**, **modular**, **settings-based and event-driven artificial intelligence** (AI) system, saving you the time to build a generic AI system (AIS) for many projects to come!

AiMalgam **solves** the **performance, scalability** and **maintenance issues** of common AI patterns, such as **finite state-machines** (FSM), **decision trees**, **behavior trees** and **rule-based systems** (RBS) and those AI asset packs that implement those patterns. The stated issues persist mainly in convoluting the decision-making process with the Action execution (forced tree traversal), rather than separating those tasks and saving performance!

If you want to read more into those issues and how these claims are proven, checkout this research paper that is the foundation of this package. The names of the evaluated asset packs have been obfuscated in the linked paper to keep them as anonymous as possible.
Disclaimer: The author of this paper, is also the developer of AiMalgam!

Note, that creating your own new AI logic via the Control Panel's AI script generator or independent of it requires at least intermediate programming knowledge in C#.

1.1 Scalability

Supported Unity versions: 2019.4 and ongoing.

Every render pipeline is supported, although actions are required for the demo scenes that include materials with Unity's built-in shader system (either convert them by hand or locate the respective render pipeline package, as it is explained later in **Section 2.1**). Note, that this is independent of the core AIS and its scripts.

Every **operating system** that **is supported** by Unity, is also supported by this AIS!

No forced pre-defined setups such as a mesh, rig or animation controller are required to run your custom AI!

New custom AI behavior can be added while working in the Unity project by offering expansion of the core AIS modules via inheritance. This includes implementing custom decision-making, Action execution and data containers, able to interact with the (modular) core AIS. This offers you the ability to build AI of any game genre, including but not limited to:

- FPS AI
- Rouge like AI
- Strategy AI
- RPG AI
- Tower Defense AI
- Platformer AI
- Stealth AI
- Survival AI
- Flying AI

1.2 Performance

Minimizing performance costs as much as technically possible is a crucial topic for game development in general, hence the integrated game AIS must not eat up too many hardware resources.

AiMalgam does not require tree traversal each time an Action has to be run and thus can save a lot of performance by providing a modular and event-driven architecture (EDA) to make decisions independent of Action calls.

This pack combines the best of the 3 main AI patterns (hence the name AiMalgam):

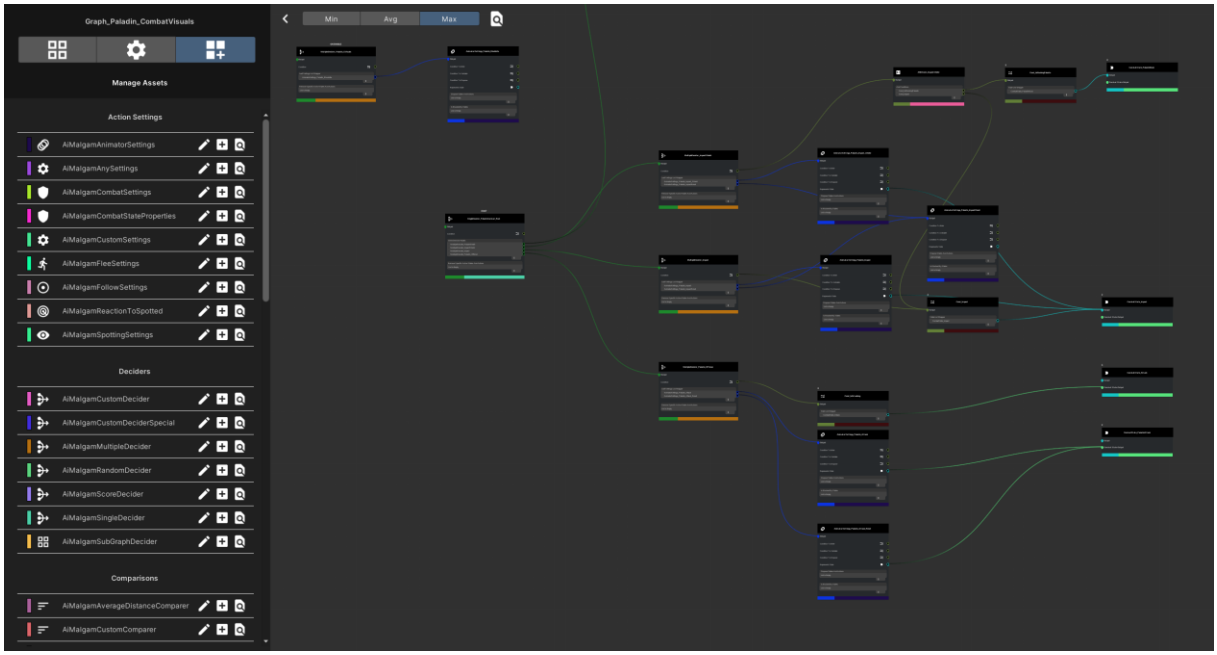
- **Blackboards** for data management and injection (per behavior, not globally)
- **Scheduling** Action executions independent of decision-making
- A mixture of the **behavior tree** and **decision tree** pattern for **decision-making only**
    - Optionally providing **State** settings to define and locate the current running Actions, as well as to identify what decisions were made in the past

If you absolutely require the traversal of decision trees each frame and not on an EDA basis, then you might want to consider using another AIS that is optimized for that use case (with less overhead).

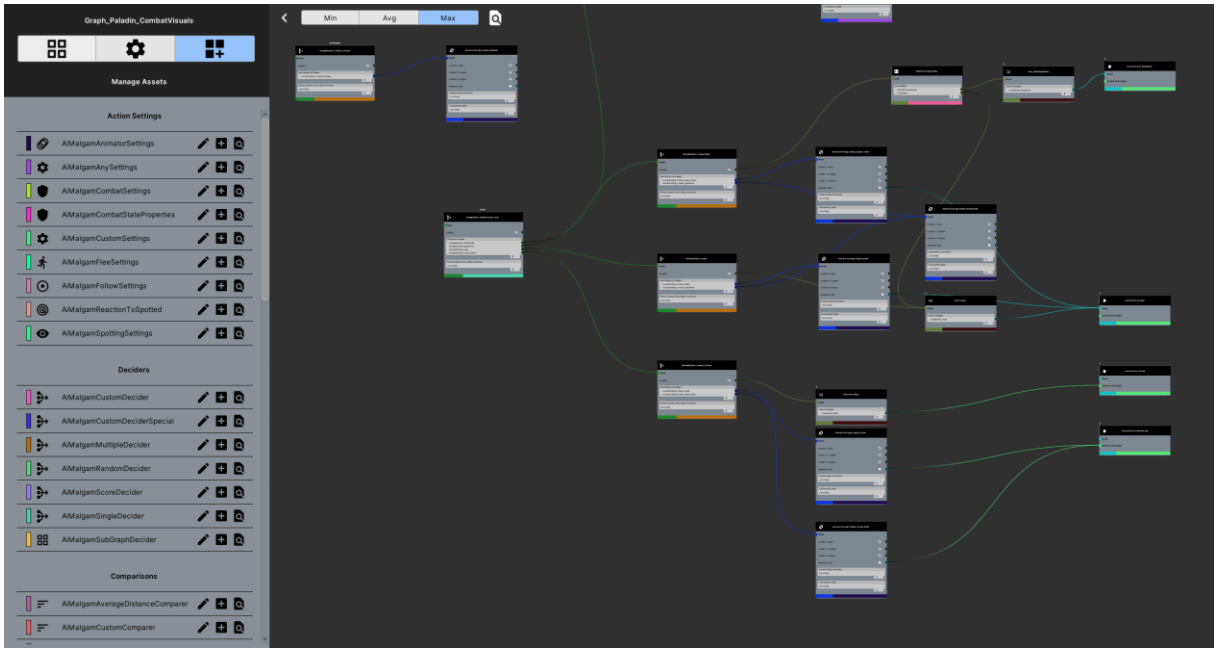
1.3 Maintainability

Building complex AI will get difficult to overview, understand and maintain over time. That's why AiMalgam provides you the node graph editor, where you can build and oversee your custom AI behavior!

Dark mode (professional license & newer Unity versions):



Light mode (personal license & older Unity versions):



You can reuse any node in multiple graphs and split your graph into further sub graph for a better AI architecture and modularity!

The graph editor mechanics are explained in **Section 2.4** and the different node setting types defining AI behavior in **Section 2.5**. To create custom node setting scripts, you can visit **Section 2.6.2**.

Also check out the video tutorial series!

2. Feature Showcase & How-to

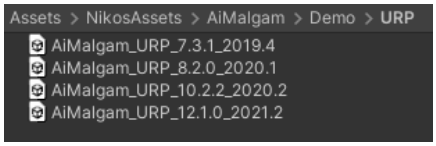
2.1 Importing and preparing AiMalgam

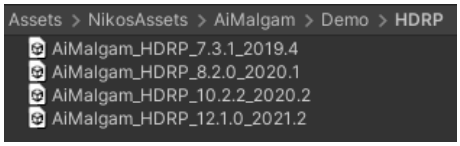
While importing the package you can either include or discard the demo section, although you might find the demo scripts very useful, since they contain example implementations for AI behavior, such as spotting, following, animation, many Conditions, et cetera. To create custom AI though, the demo section is not mandatory!

Optional: *Instead of importing the sub-pack "HelperTools" in Dependencies/HelperTools/ directly, you can also get it here or here* ***for free****. Be prepared though, to re-assign the missing assembly definition files in AiMalgam (Core & Demo) if you choose to do so.*

Including the demo and displaying the materials correctly you might need to make some changes depending on your current render pipeline. If you use the built-in render pipeline, no actions are required!

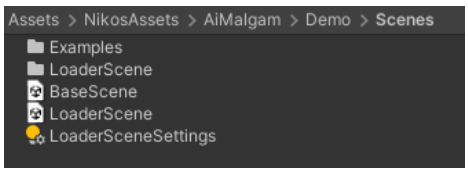
But if you use the URP or HDRP setup, you either need to convert the materials "by hand" via the render pipeline wizard (see the HDRP and URP manuals) or locate the render pipeline package in the path Demo/URP or Demo/HDRP:



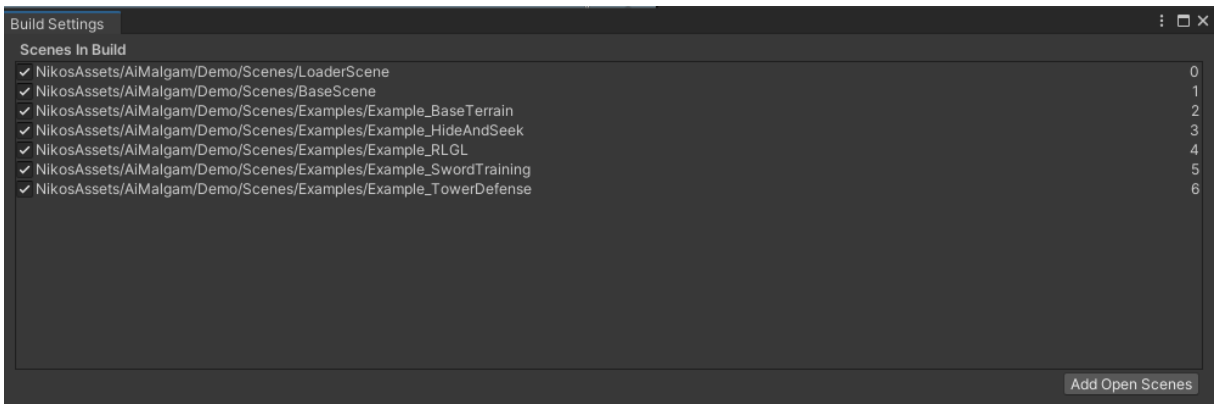


Note, that the material **color animations** for **demo hide spot entities** are **not working in** the other **pipelines**, since the shader parameter names differ from the built-in system. This is a simple animation file issue, which will be addressed in future updates.

The example scenes are located at Demo/Scenes, where you can either load the "LoaderScene" or the individual scenes at Demo/Scenes/Examples/:



To load the demo scenes dynamically, it is **very important** that you add them to your build settings:



Note, that in some Unity versions the (NavMeshAgent) movement speed of the **Demo** entities differs. You can adjust the values in the respective *MalgyNavMeshSpeedAdjustment* MonoBehaviours if necessary.

2.2 Demos

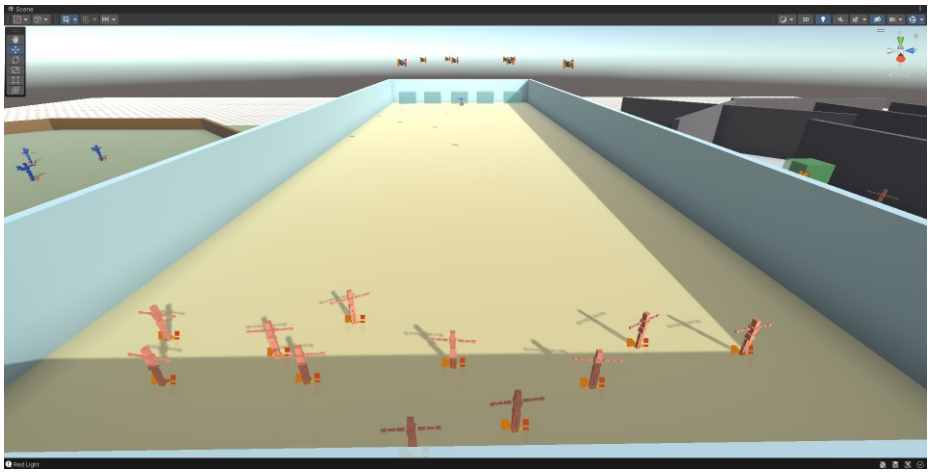
This section briefly explains the available demo examples that expand from the AiMalgam core AIS. You can also visit the video tutorial series that explain how some Demo setups work in detail.

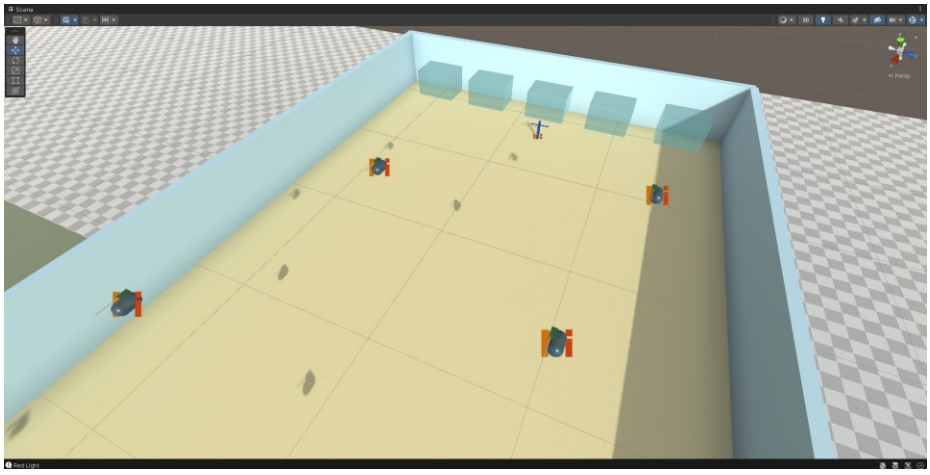
Note, that the demo setup uses quick setup models and animations to wrap the AI examples into something observable. You can create or use any **AAA** (or **low poly**) models, rigs, animations, textures you like to work with your custom AI created with this asset pack!

Also, the presented AI behavior implementations do not have to be used. You can create your own custom AI logic as explained in **Section 2.6.2**.

2.2.1 Red Light Green Light (Similar to "Squid Game")

This demo sample showcases a simple version of the red light green light game similar to the "Squid Game" series on Netflix:



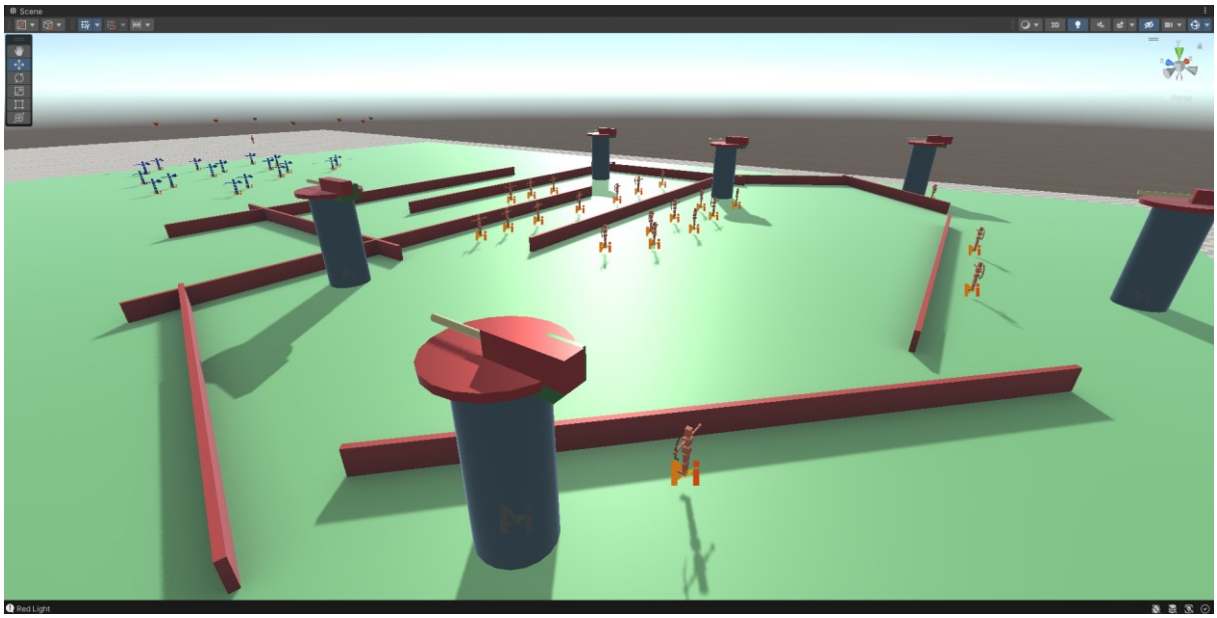


The red entities represent the game participants who try to reach the blue trigger boxes without dying. The blue entity is the observer and spots early runners or slow stoppers, which then get shot by the drone entity that is flying above. As you can see in the drone example you can create AI that does not use any rig or animator controller and is thus not bound to any pre-defined or forced upon setups!

The (red) runners die when shot and play a death animation before vanishing from the world. For each death, a new runner spawns and gets added **DYNAMICALLY** to the scene!
So, there are no individual or "hard references" between the observer, drones and runners.

2.2.2 Tower Defense

The tower defense setup combines multiple AI behaviors and different entities together, again showing you that there are no borders for funky or different entities!



We have the archer, who prefers distance-based combat but can also strike with melee hits before attempting to flee at some point in time within a restricted area.

The tower that remains in place and can only shoot projectiles, similar to the drone attacker.

The drone attacker, which slowly creeps from above targeting the closest target and shooting at it until its gone.

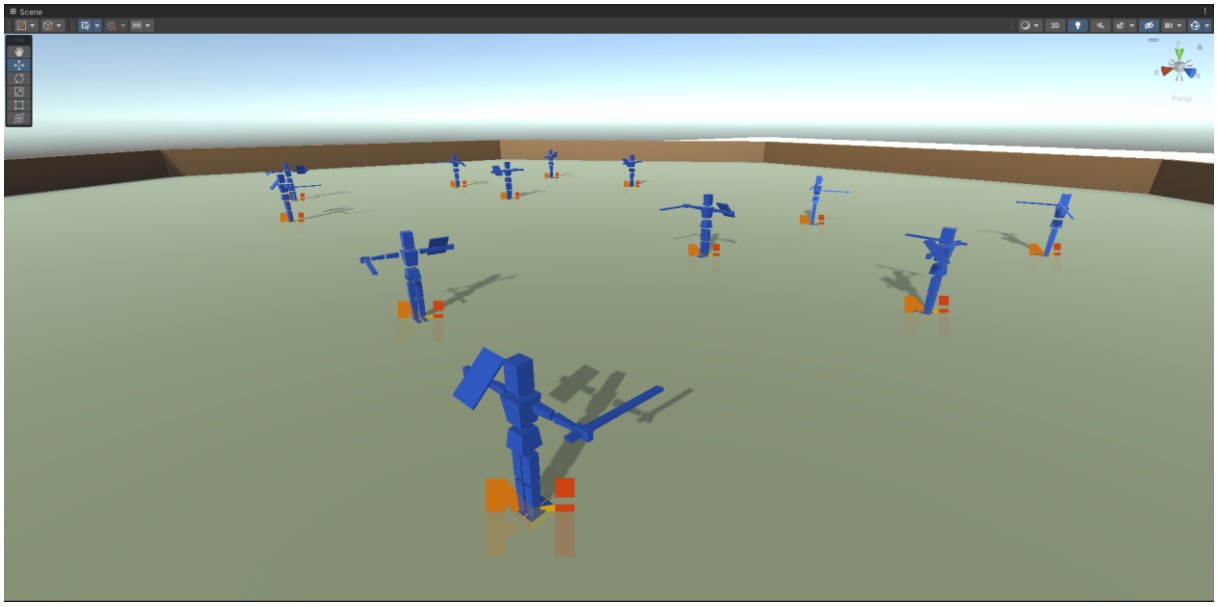
And finally, the paladin who can only do melee combat, also chasing after the closest target. The paladin will respawn again and again, so the defenses have no chance in winning sadly…

Every humanoid entity in this example can react to hits and play a random hit animation, interrupting its combat.

2.2.3 Sword Training

Talking about hit impact reactions, this scene is providing yet another example on that topic.

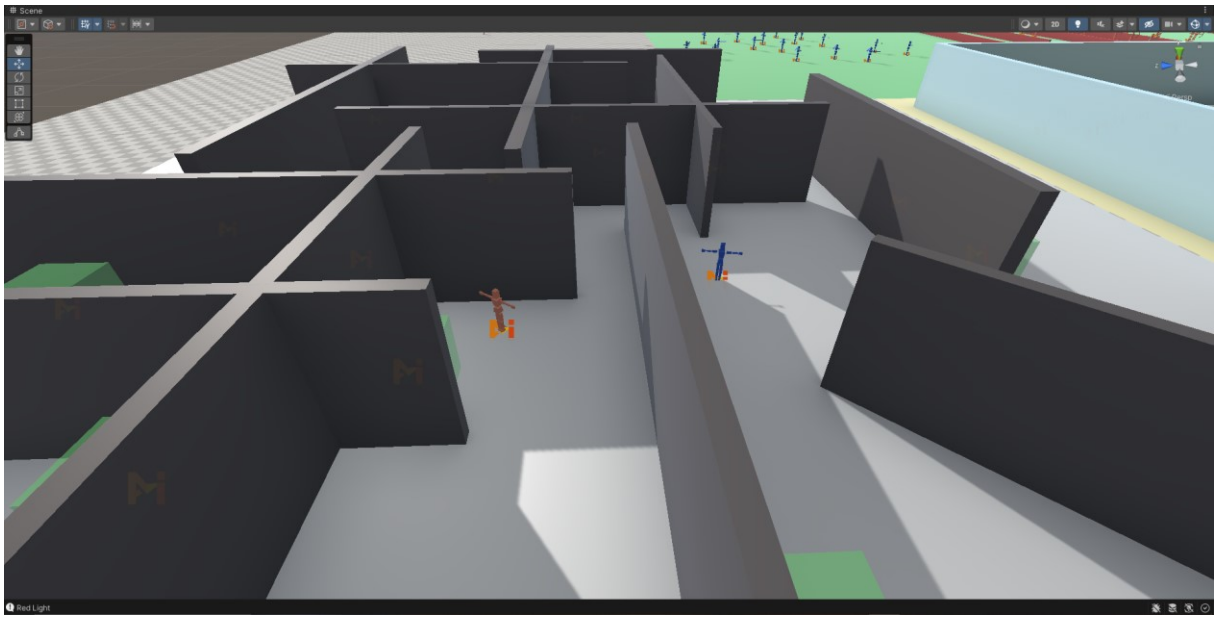
Here a few paladins fight each other (no harm done) and react on impacts either by blocking or playing an impact animation, depending on their Combat-State. This shows that you can setup AI with more advanced combat mechanics, such as evading, parrying or blocking an attack.



Since the animations and animation controllers need to be refined further, in addition to some trigger hit events not being called, some impacts might be missed out (Demo will be improved in the upcoming AiMalgam versions).
Also, when building your own animation combat system, make sure to sync your animations with your combat tools and systems. Depending on your setup, this will require some fine tuning and time (see this video)!

2.2.4 Hide & Seek

Lastly, a simpler setup, displaying entities that try to hide in certain hide spot boxes and seekers that try to find and chase the hiders:



Fun fact, the hide boxes also represent AI entities! They contain a node graph that lets them change colors by playing an animation (only in the built-in render pipeline) depending on if a seeker or a hider entered the trigger box.

The hiders search the place for empty hide spots, go into one if found or flee when being spotted by a seeker.

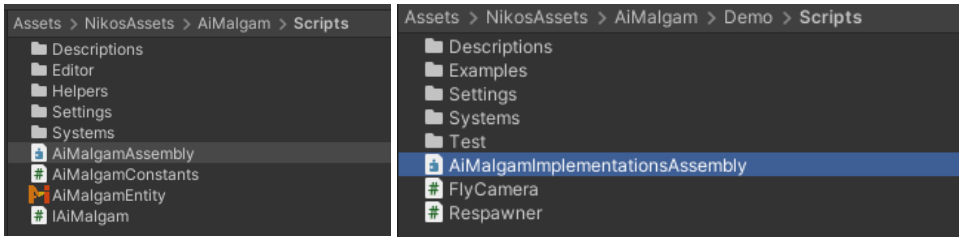
Similarly, the seeker searches for hide spots to reveal them and chases hiders when found, until the hider is out of sight (and memory – about 2-3 seconds).

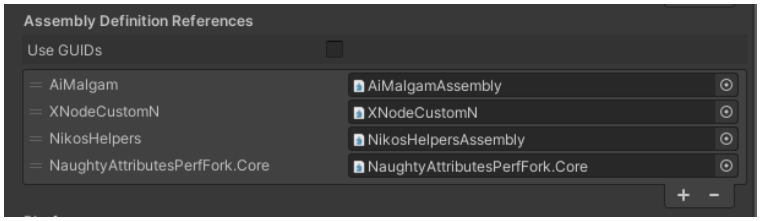
2.3 API Documentation

You can find the AiMalgam core API documentation here and the demo code documentation here. The dependency documentations can be accessed here and here.

If you work with assembly definition files for your scripts and want to access and make use of the AiMalgam code base, make sure to reference its assembly definition files:

- AiMalgamAssembly.asmdef
- AiMalgamImplementationsAssembly.asmdef

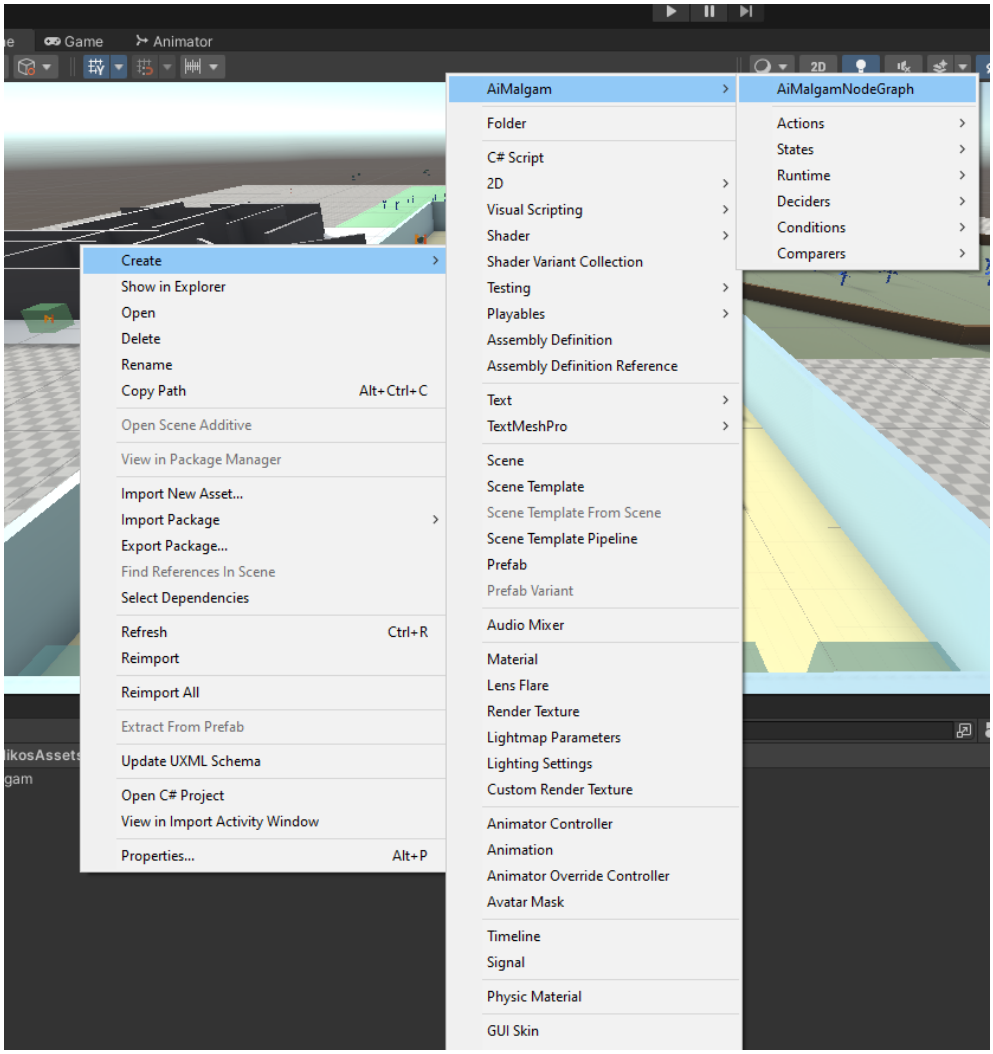




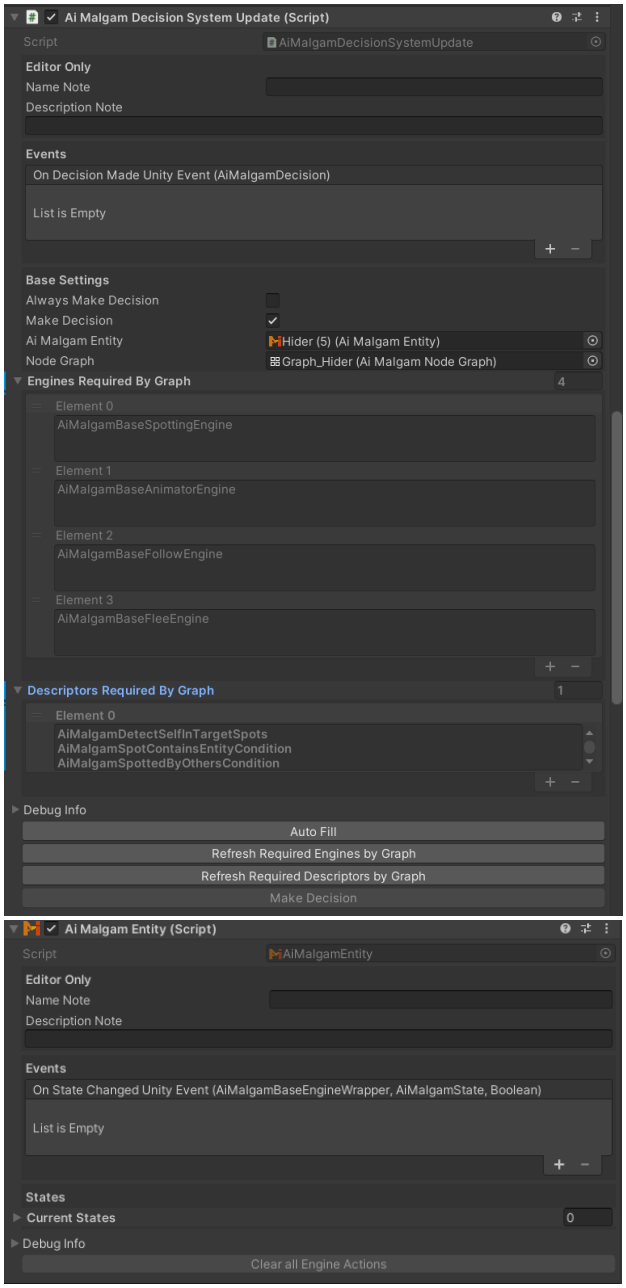
2.4 Working with the Graph Editor

It is recommended to watch the video tutorial series in addition to reading this manual for better understanding!

You can add a new AiMalgamNodeGraph by opening the context menu in your desired location:



To run a node graph on an AiMalgamEntity, you need to add the AiMalgamDecisionSystem (see API) MonoBehaviour to a GameObject and assign the respective graph to the “Node Graph” field, as well as the “AiMalgamEntity” field:



To run the NodeGraph’s Action settings, you also need to add the required Engine (Action scheduler) MonoBehaviour that can be viewed in the “EnginesRequiredByGraph” list. Make sure to hit the “Refresh Required Engines by Graph” button at the bottom to update the list. See this video for Engine implementation examples.

The same goes for the Descriptor MonoBehaviours required by special Condition node settings present in the NodeGraph. You can view them in the “DescriptorsRequiredByGraph” list and also refresh them by pressing the “Refresh Required Descriptors by Graph” button.

The decision-making process is only called, if the “Make Decision” or the **“**Always Make Decision” fields are checked, or if you call the “MakeDecision()” method directly (see API). If you use a custom Decision-System that doesn’t call the “Tick()” method, no decisions can be made even though the check boxes are ticked.
In this case though (AiMalgamDecisionSystemUpdate), the “Tick()” method is called each Update frame.

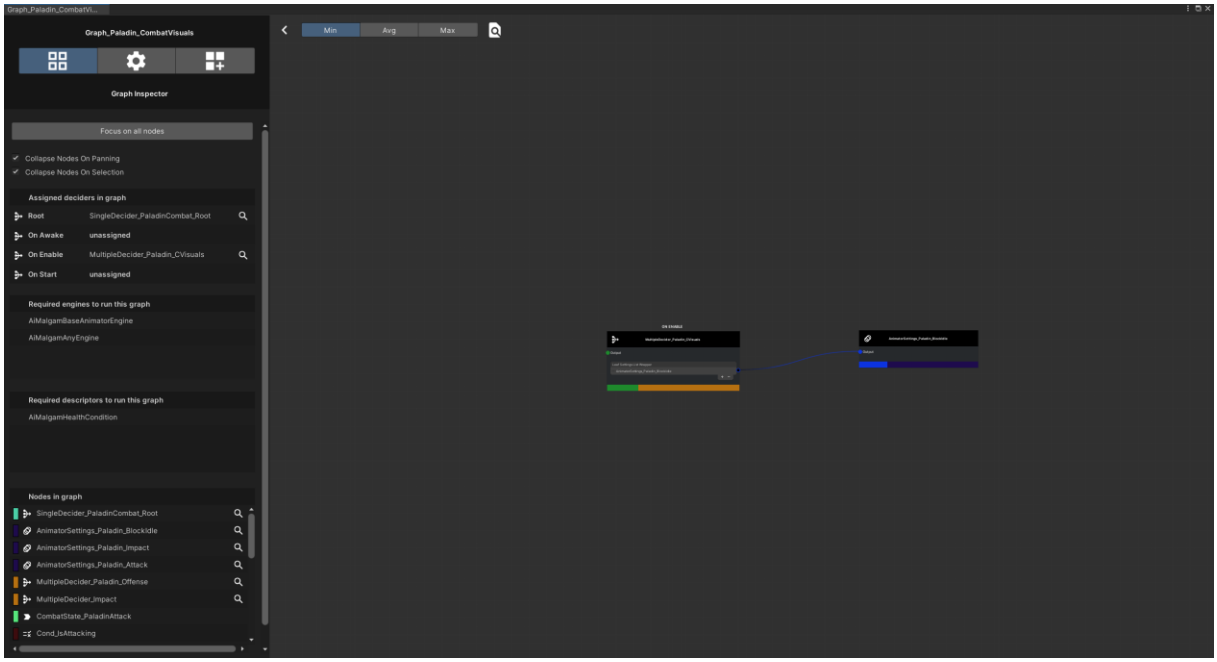
It is generally advised to trigger a decision once an event occurred (spotting, hit, destination reached, etc.) and not each frame to increase the performance of your game. Again, calling the “Tick()” method each frame **does not trigger a decision** automatically, but rather times the next possible decision you triggered via its described fields or “MakeDecision()” method!

View the diagrams for Decision-Systems in **Section 2.5.2**!

The Graph Editor

On the left-hand side of the graph editor, you find the side-panel with 3 sub areas you can access by clicking on their respective tab in the header. Said tabs help you to configure and display information about the currently opened graph, access a node Inspector that supports editing of multiple nodes of same type at once and create new node settings via the manage assets tab.

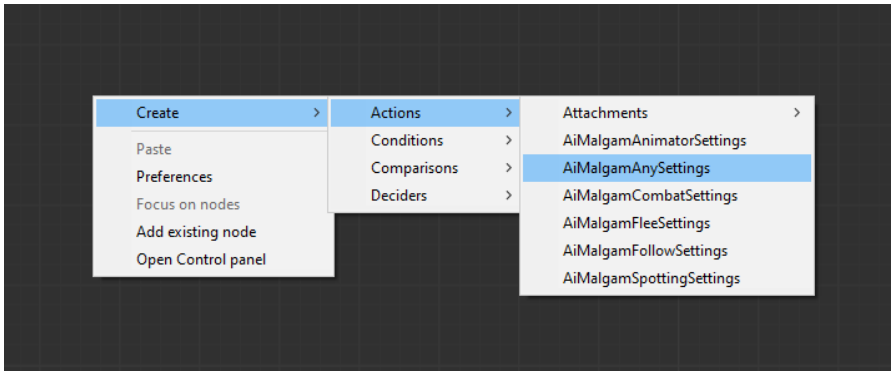
The floating mini tabs to the right of the side panel help you to toggle the complexity and node visibility in your graph. It is recommended to work on “Min” or “Avg” display to have a cleaner overview and also save rendering performance.
The small search icon lets you find the graph ScriptableObject in your project window by selecting it.



You can pan the grid are by holding and dragging either the right or middle mouse button. By scrolling with the mouse wheel, you can also zoom in or out of the grid-view.

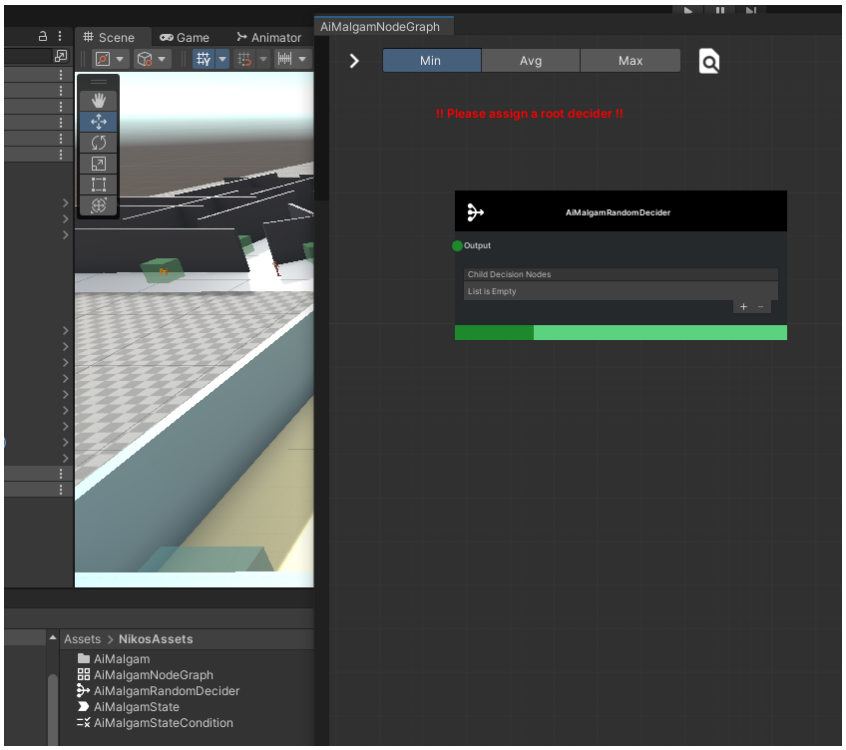
The Grid Area

Add nodes by either dragging them from the project window or using the context menu on the grid area:

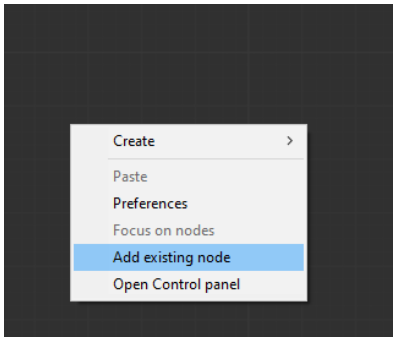


Note, that you should **not create** or **connect** nodes **at runtime** to prevent issues and unexpected AI behavior! Changing the settings of a node though is absolutely fine!

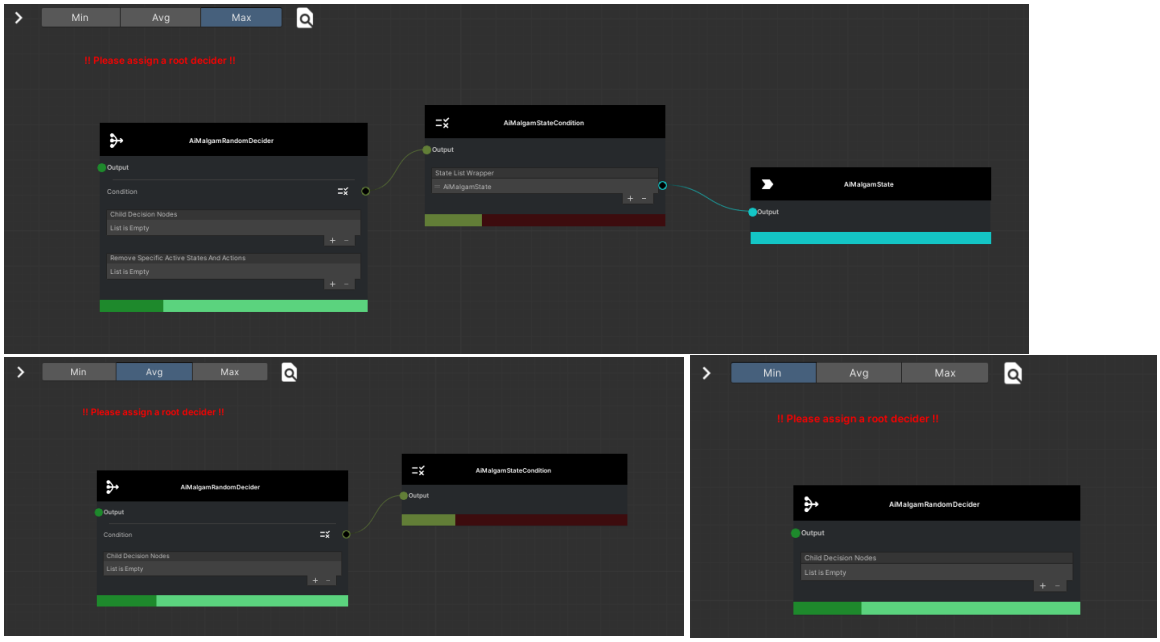
The newly created node's path is by default the same folder, where the graph editor is located as well, unless you set a different location in the manage assets side panel:



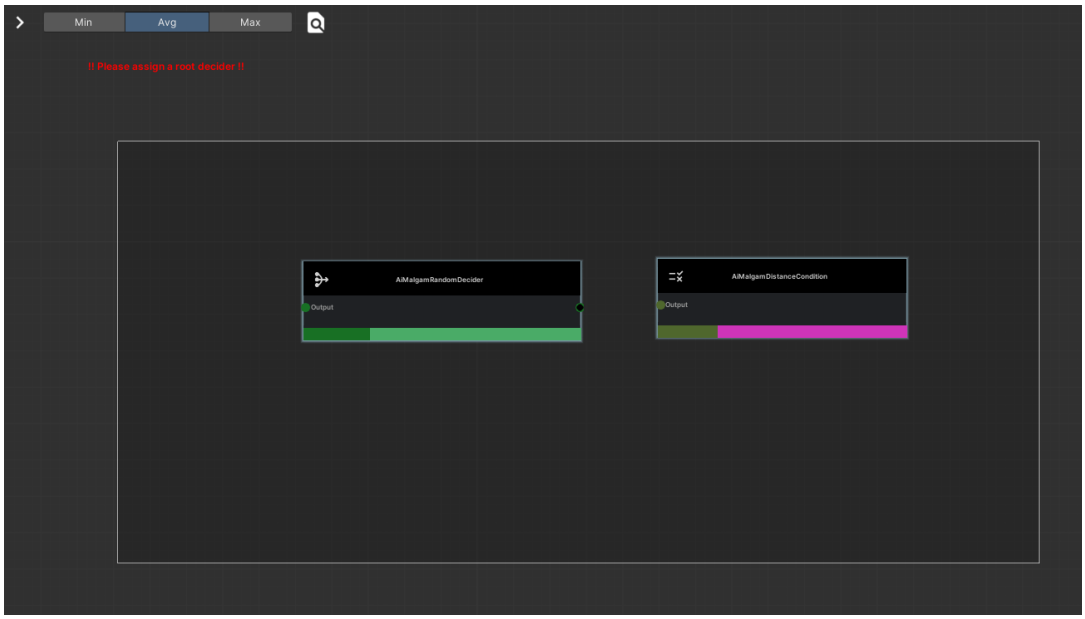
You can also add existing nodes and reuse them in other graphs. Be careful though, since changing the node's settings will take effect everywhere it is referenced:



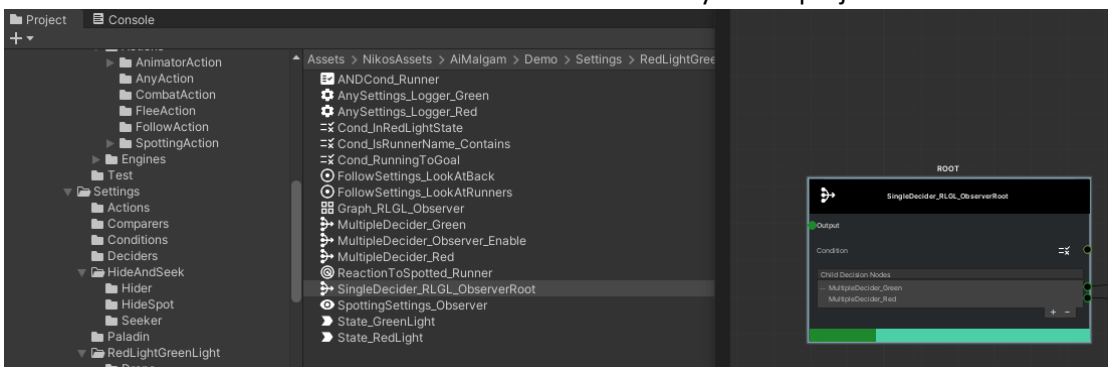
Note that if you drag nodes, they or their node connections might not appear in the graph if the graph's visibility is lower than the dragged node's visibility. The nodes and their connections do still exist but are simply not rendered:



You can select nodes individually by pressing CMD on Mac and ctrl or shift on windows or select multiple with a selection grid by holding and dragging the left mouse button on an empty grid-area:



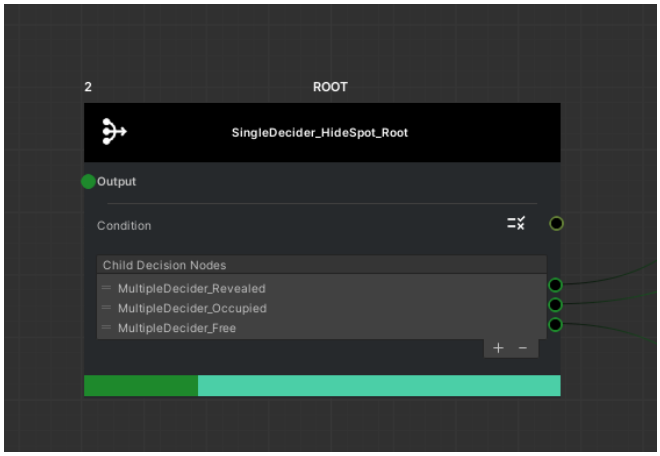
Selected nodes will also be selected and shown in the Unity Editor project view:



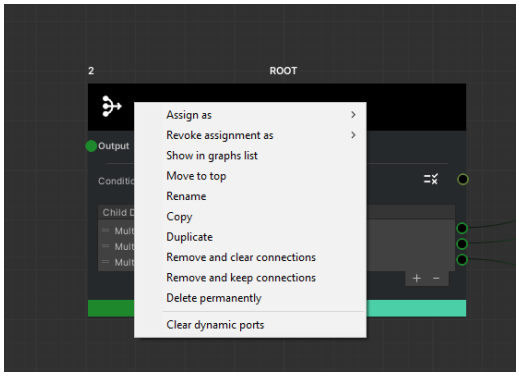
The Node Body

The node consists of 4 areas:

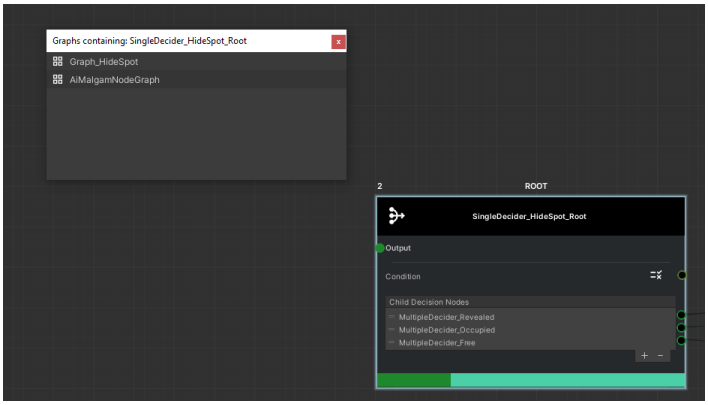
- **The floating header**
  - Contains a small number on the left corner, indicating how many graphs host this node setting. If not existent, this is the only graph (1 total graphs).
  - The central area defines the special Decider assignment for this node (Only visible and accessible for Decider node settings). This indicates from where to traverse the decision-tree and when. The diagram found in **Section 2.5.2** illustrates this in detail.
- **The solid (black) header**
  - Contains an optional icon of the node in the left corner and displays the node name in the middle.
- **The node body**
  - Displays output and input ports of the node, which can only reference other AiMalgamNodes in a single field or ListWrapper (see **Section 2.6.3** and this video). No primitive or other data types are displayed here, but you can access them in any Unity Inspector.
- **The node footer**
  - Displays the base type color on the left-hand side and the special (inherited color) beside it. This helps you to find compatible nodes for other ports or spot and identify the node setting type in the graph editor generally.



Opening the context menu while hovering over the node gives you yet another options-list:



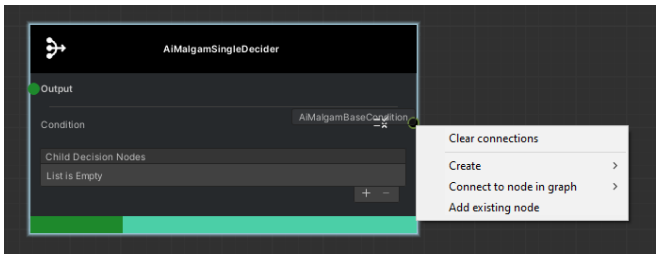
- **Assignment and Revoke assignment** are only visible on Decider nodes and help you to configure how and when a Decider should be traversed by its calling Decision-System MonoBehaviour. One Decider can have multiple assignments, but the assignments cannot be shared among other Deciders.
- **Show in graphs list** will open a popup window displaying a list of graphs that host this exact node you selected:

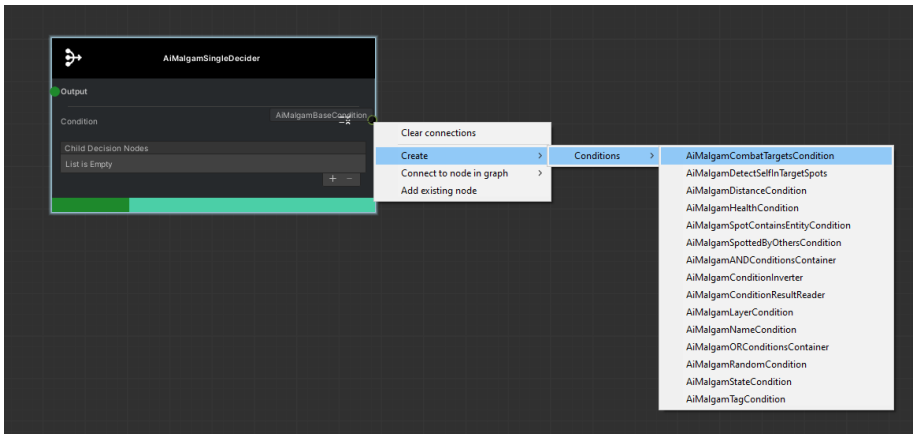
  By selecting a graph from the popup list, the graph object itself will be located and selected in the Unity Editor (project window).
- **Move to top** will render this node above all other nodes.
- **Rename the node**. Don't be shy to also rename the file itself in the project window!
- **Copying or duplicating this node** will also copy the **input** (not output) port connections of the original node. Also note, that the file location of the node is the same as the original copied one.
- **Removing a node** while keeping the ScriptableObject still in the project
  - **Removing and clearing the connections** will cut any port connection to nodes that remained in the graph. It will keep connections to those nodes, you have removed with it via group selection.
  - **Removing and keeping its connections** will still keep input and output port connections to nodes that remained in the graph. Note that if you reopen the graph, the node will reappear if its **output** port was connected to an **input** port of a remained node (but not the other way around).
- **Deleting the node** permanently will also wipe the ScriptableObject, which cannot be recovered! Also, the **Undo/Redo history** will be cleared to avoid reference and port connection issues.
- **Clearing dynamic ports** will clear and unset every list item input port of this node.

The Node Port

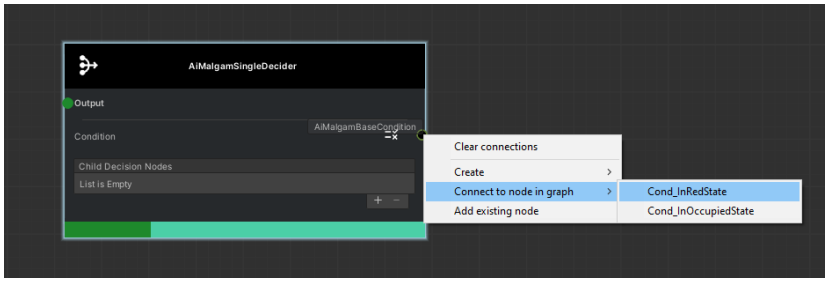
Hovering and opening the context menu over an input port will display additional options, like breaking an existing connection or creating a new one:



Create and auto connect compatible nodes to the hovered port:

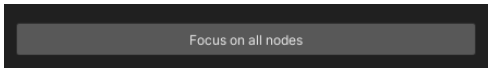


Auto connect to compatible nodes in the same graph, without dragging the other desired output port (across a long distance) to the hovered port:

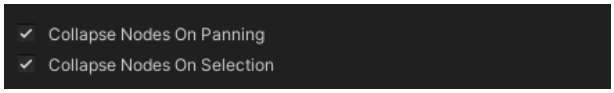


The Graph's Side Panel [Graph Inspector Tab]

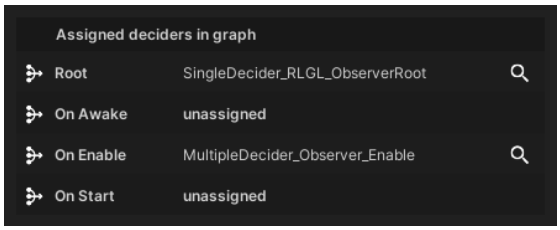
If you ever get lost in the grid-area and can't find your nodes, press this button:



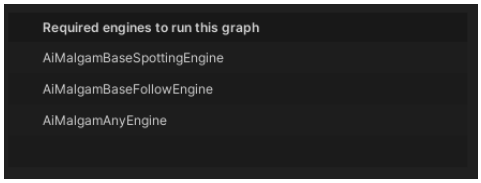
By default, nodes collapse on panning the grid-area and selection. You can turn those off with the following checkboxes:



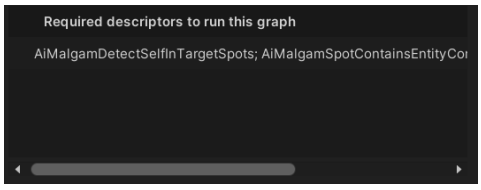
View and select special assigned Deciders in the graph by clicking on the list item or the small search icon on the right of each Decider to focus it as well:



View the required MonoBehaviour Engines to run all Action settings on the target AiMalgamEntity that are present in this graph (same base types occupy the same row):



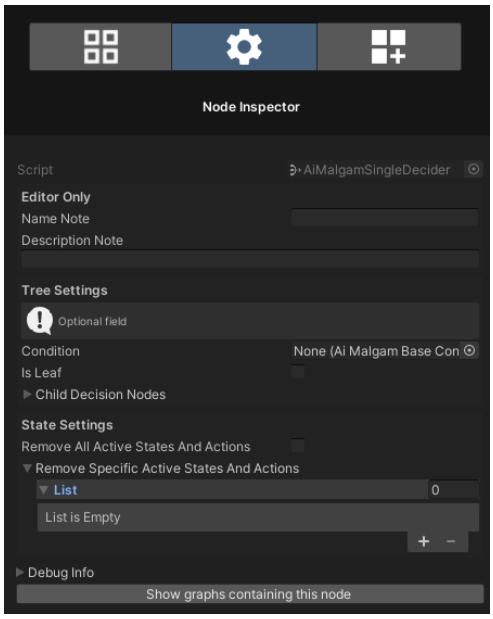
View the required MonoBehaviour Descriptors to run special Condition settings on the target AiMalgamEntity that are present in this graph (same base types occupy the same row):



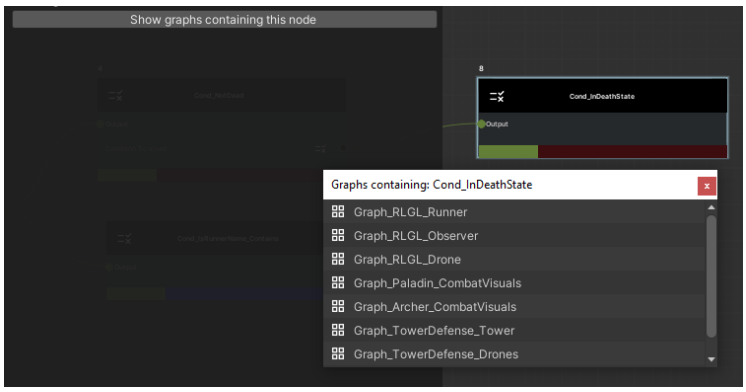
View, select and focus node settings that are present in the current graph. Hidden nodes cannot be focused (search icon is hidden as well) but can still be selected and configured:

The Graph's Side Panel [Node Inspector Tab]

Edit node settings, just like in any other Unity Editor Inspector window:



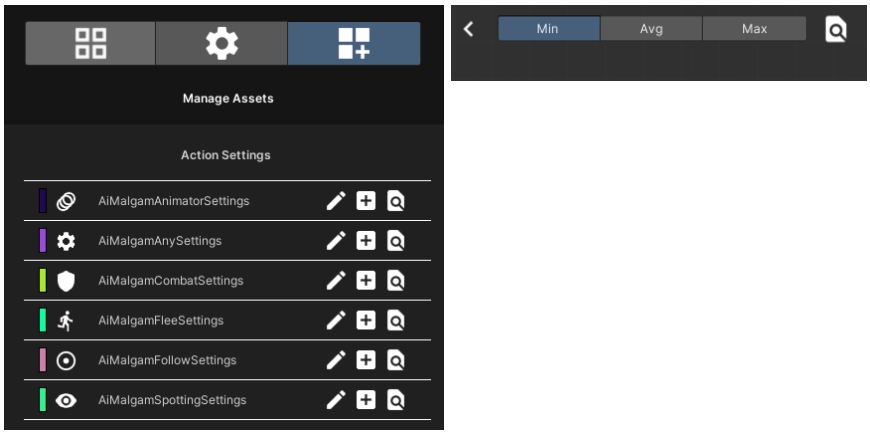
The bottom-most button reveals a popup that shows a list of graphs that host the same selected node:



Selecting one graph list item will find and select it in the Unity Editor project window as well!

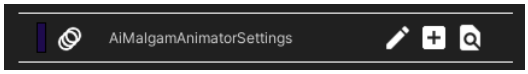
The Graph's Side Panel [Manage Assets Tab]

View, add existing, configure the ScriptableObject's creation path or create new node settings. Depending on the node visibility, some setting sections might not appear:

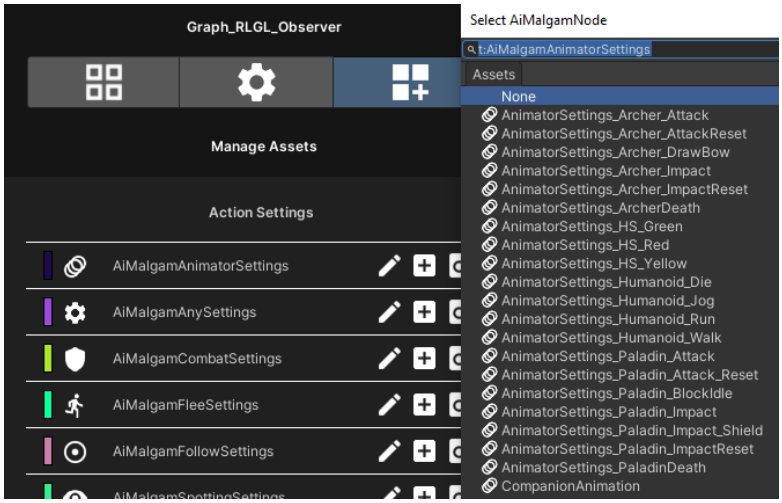


With the **edit icon** (pen-shaped button) you can choose the **file path** for the **next** created **node** setting that is created from this graph (also via context menues). By default the node's file path is the same as the graph's one. The only **exception** to this **is cloning or copying nodes**. Those nodes will have the same path, as the originally cloned node.

Pressing the "+" button, creates a new node setting of the chosen type in the setup path:



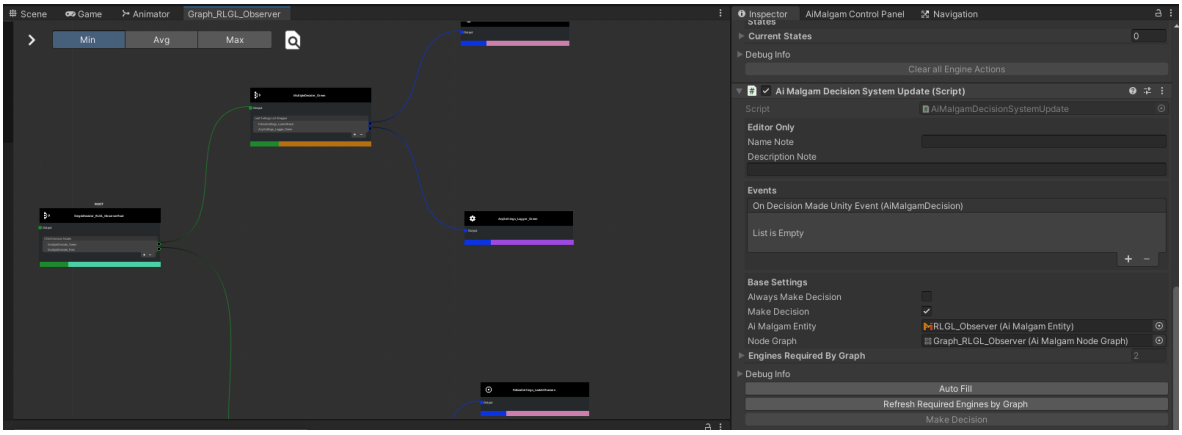
The **search icon** button will open a list of node settings (of chosen type) that already exist in the project and can be added to this graph as well:



Debugging AI Behavior

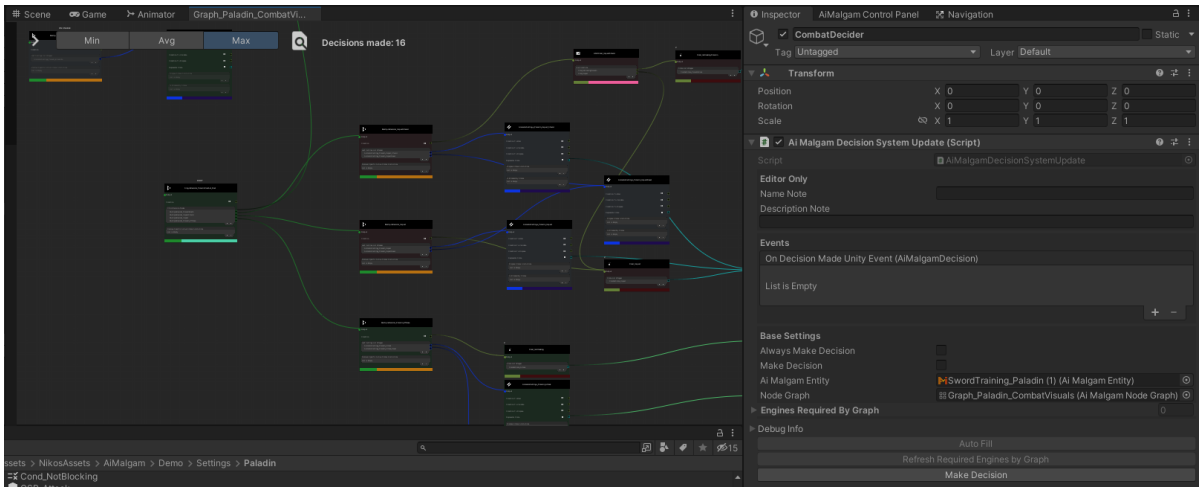
Debugging the graph editor only works in the Unity Editor and while in play mode!

First, you need to open the graph you want to debug, then you need to select the Decision-System MonoBehaviour (or its hosting GameObject) that either references the opened graph directly, or one of its Subgraph-Deciders in it. Do not confuse the Decision-System Component with the AiMalgamEntity Component:

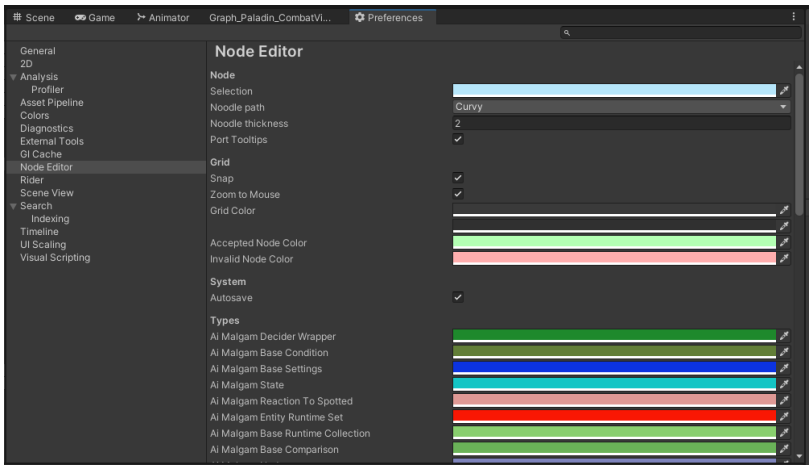


While in play mode you will see colored nodes (**green** = traversed and valid, **red** = traversed and invalid, default/white = not traversed), also shown in this video. The marked nodes will be reset before each decision-cycle, so keep in mind that your custom marked nodes might be reset if you handle the marking outside of the decision-making cycle/ timing.
See the life cycle diagram in **Section 2.5.2**, the API, or the video series to know when to mark your own node settings in the graph.

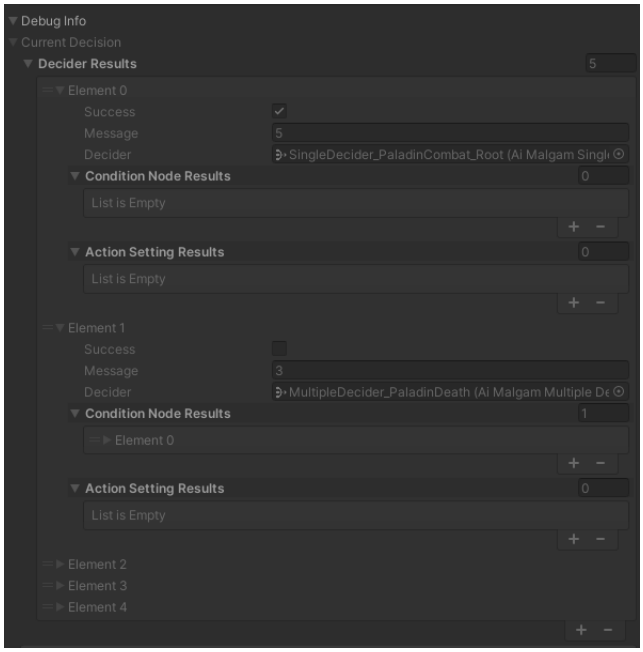
On the top left you will notice a number showing up, which displays the current decisions made by the selected Decision-System. This helps you to find out if the decision-making process is called too often or not often enough:



You can change the colors in the node preferences window. Make sure to reopen the graph after the changes are made:



Unfolding the MonoBehaviour's "Debug Info" foldout and "Current Decision" field, will show a flat list of traversed Deciders and Conditions:



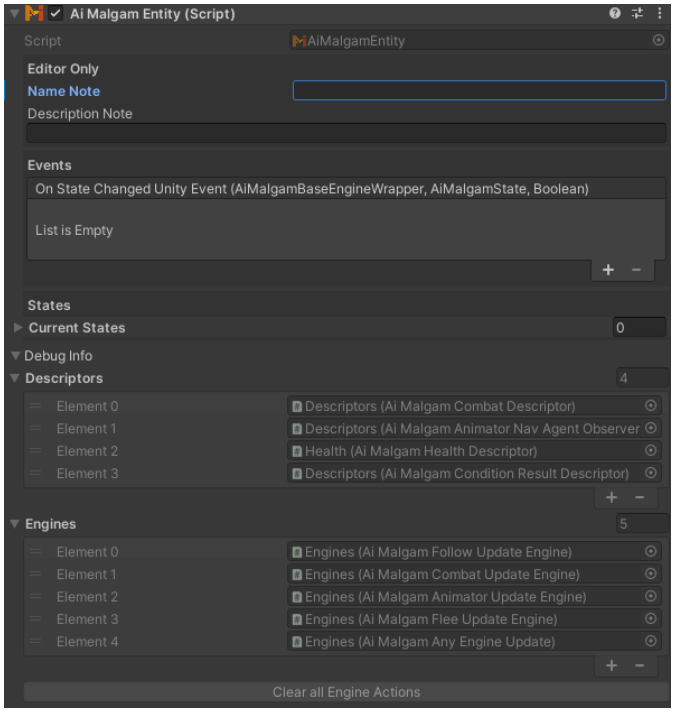
2.5 Understanding the AI Workflow and what Node Settings to use

See this instruction video for this topic.

2.5.1 AiMalgamEntity

See the API here.

The central AI MonoBehaviour containing references to Descriptors (custom data container MonoBehaviours) and Engines (Action schedulers):



The entity is crucial for decision-making and running Actions. It is the main required Component to represent an AI entity instance in the runtime game environment.

You can acquire the required Engines (see **Section 2.5.6**) or Descriptors (see **Section 2.5.7**) by the respective get methods or from the list directly. Those methods are used to assign Action settings to their compatible Engines or get Descriptors for Condition settings, all without using expensive C# reflection calls but rather pre-generated hash values.

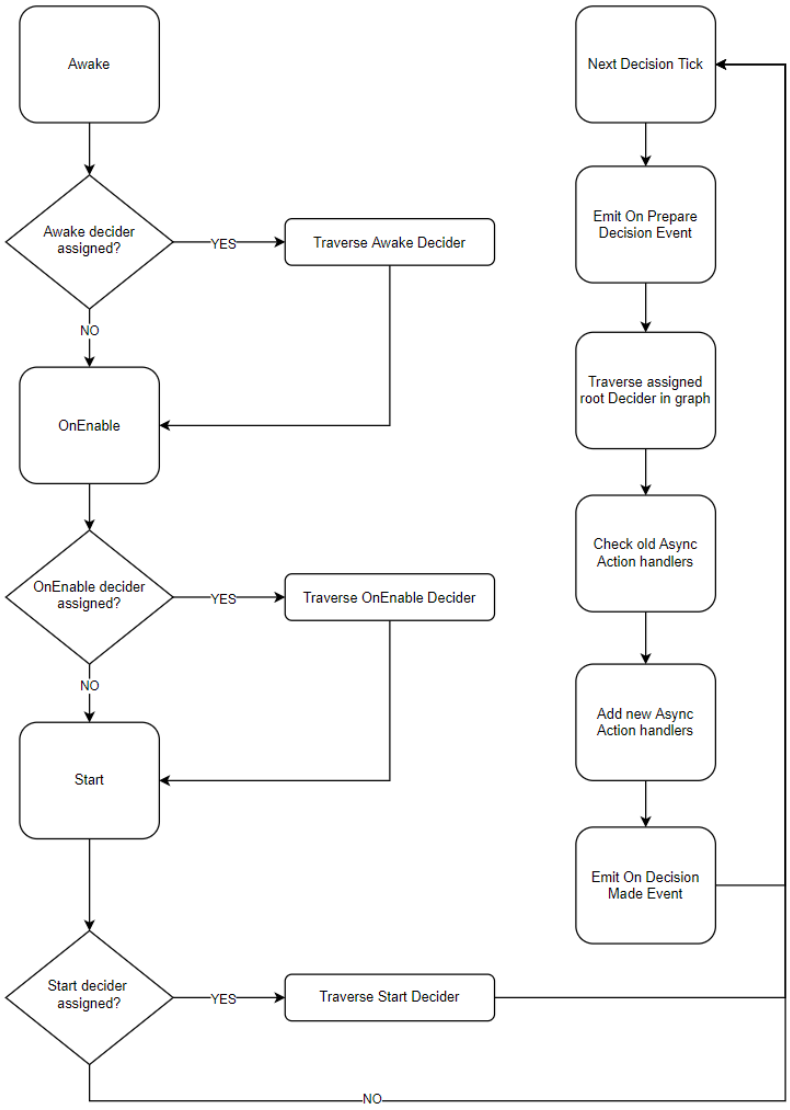
The Descriptor and Engine MonoBehaviours store the hash values in the “Debug Info” foldout. They are generated automatically upon initialization.

2.5.2 Decision-Systems

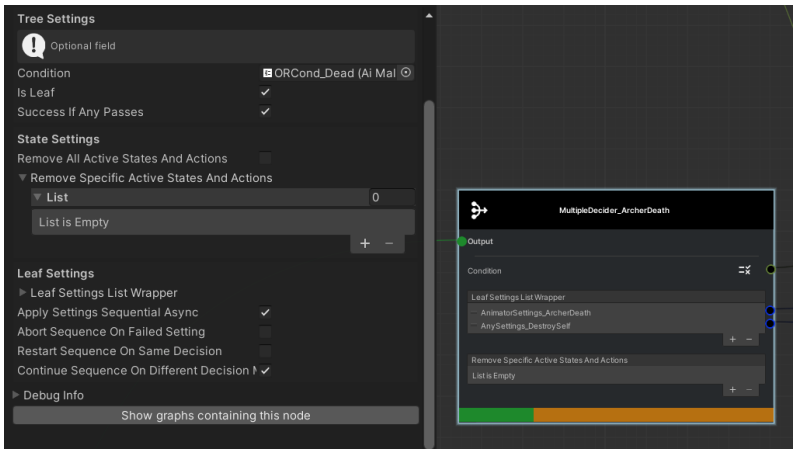
See the API here.

Section 2.4 already explains the purpose of the Decision-System in detail.

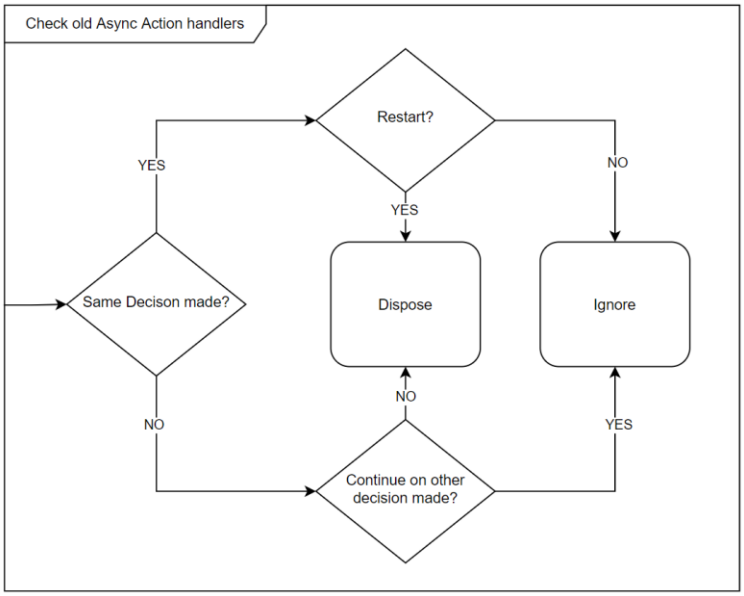
The Life Cycle and Tick Schedule of a Decision-System



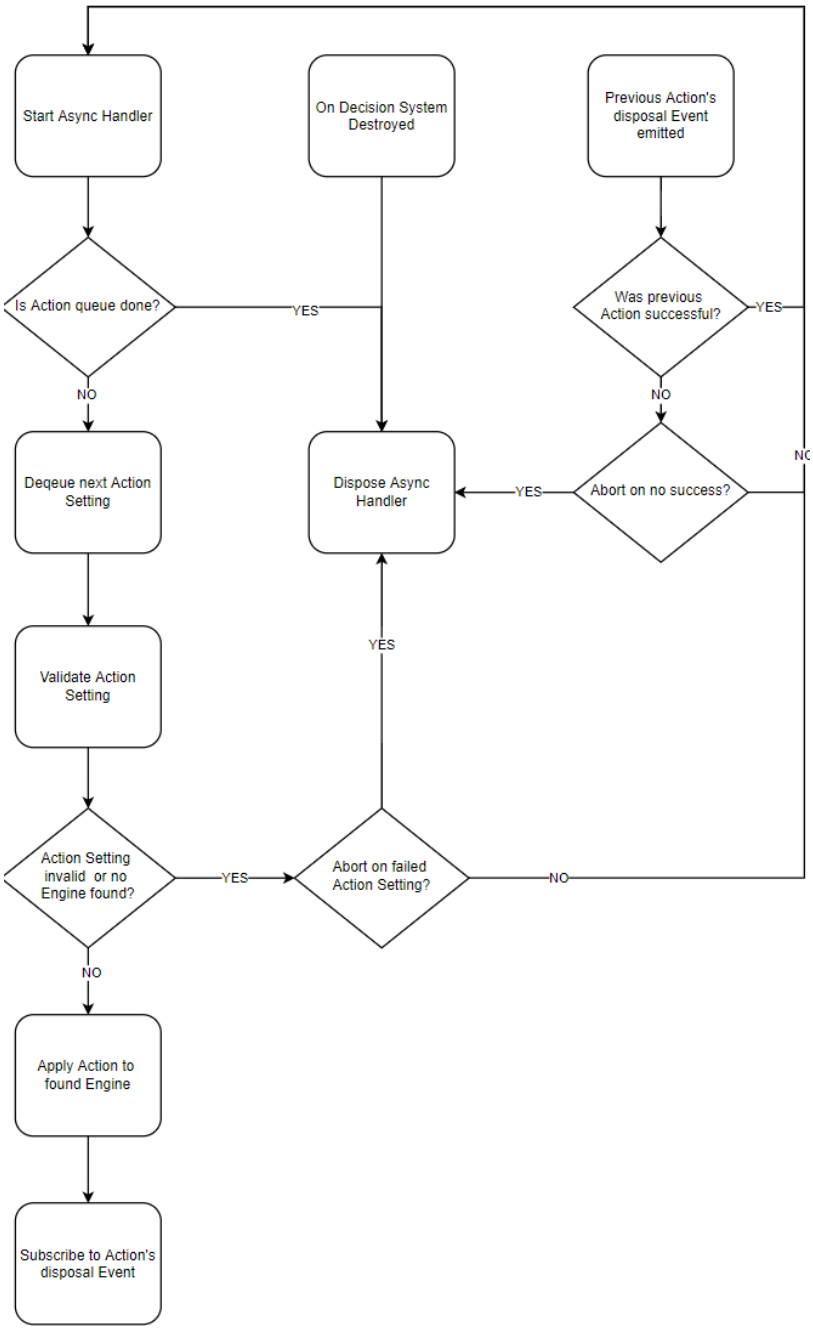
Async sequential Action handlers (see API) can be configured via the Multiple-Decider node settings:



Checking old Async Action Handlers (from Multiple-Decider Settings)



Async Action Handler Life Cycle (from Multiple-Decider Settings)



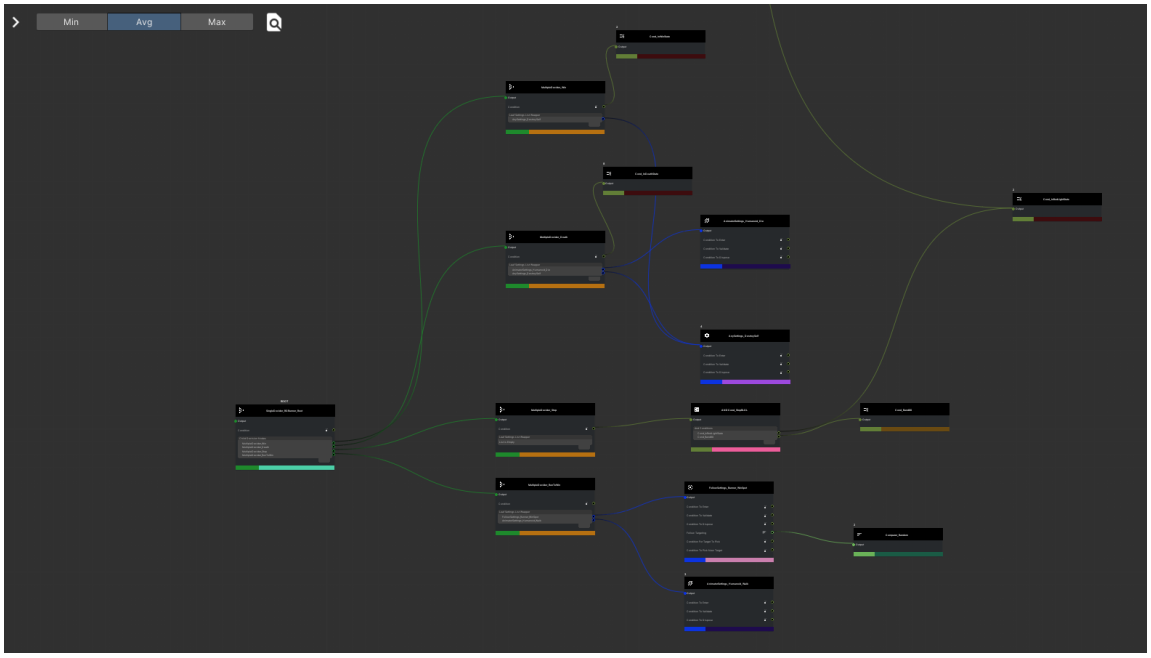
2.5.3 Settings

Any AI behavior is configured via Unity's ScriptableObject system and can be displayed and maintained in the graph editor (see **Section 2.4**).

Each AiMalgam setting inherits from the AiMalgamNode (see API) and depending on its type, covers a specific functionality for the decision-making process. You can create your own node settings as shown in the video here and later in **Section 2.6.2**.

Note, that you should **not create** nodes, **change** or make **new connections at runtime** to prevent issues and unexpected AI behavior. Changing the settings of a node though is absolutely fine!

The following illustration contains an AI decision-tree that uses Decider node settings in combination with Condition settings to branch between other Deciders and Action settings as leaves:



Section 2.4 explains that you need to set up additional Components (AiMalgamEntity & AiMalgamDecisionSystem) to run the desired graph.

Summarized, each node setting type has a special purpose for the AI flow:

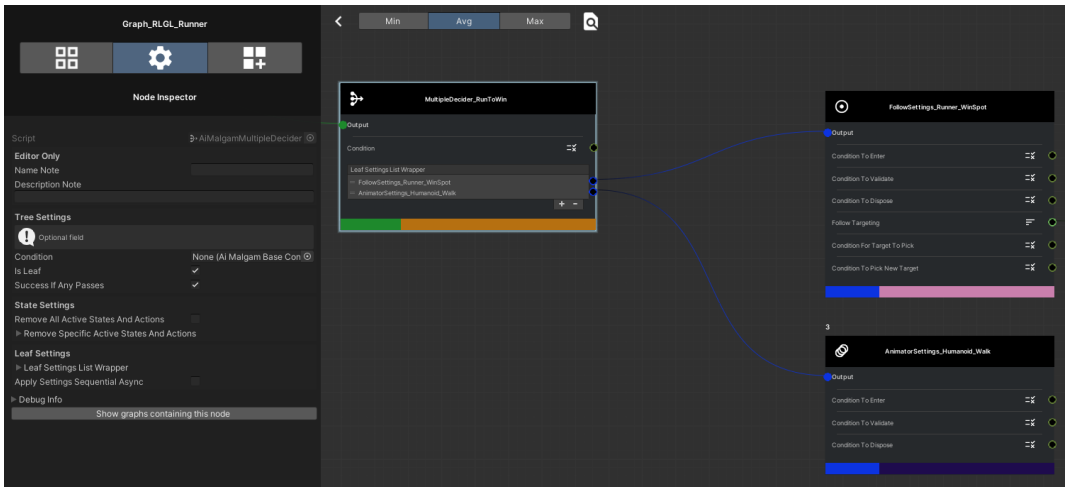
- **Deciders** branch the AI decision
- **Conditions** (in)validate (block or pass the tree traversal)
- **Comparers** are used to sort and compare a list of AiMalgamEntities
- **Action settings** represent AI **Actions** (animations, sounds, movement, basically anything that interacts with the game environment).
- **State settings** are used to find and represent Actions (also to dispose looping ones)
- **Runtime-Sets** store global (primitive) data

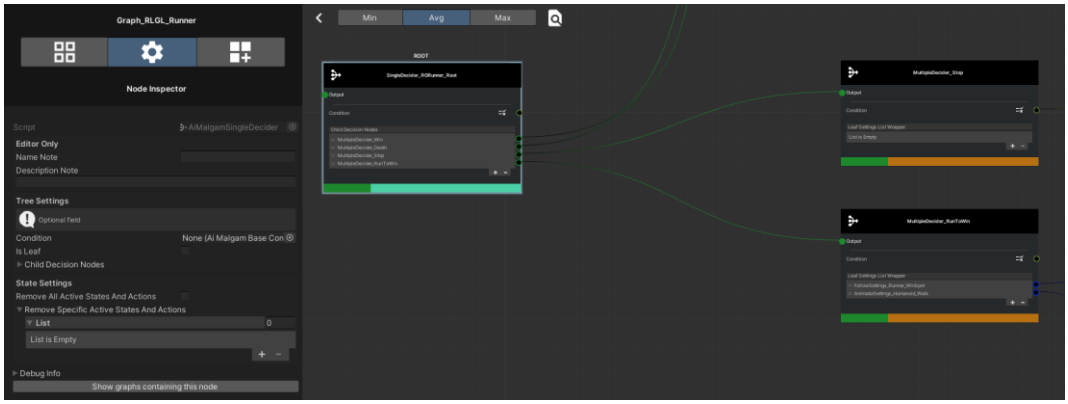
You can also create your own node types as shown in **Section 2.6.3** or in this video.

Decider Settings

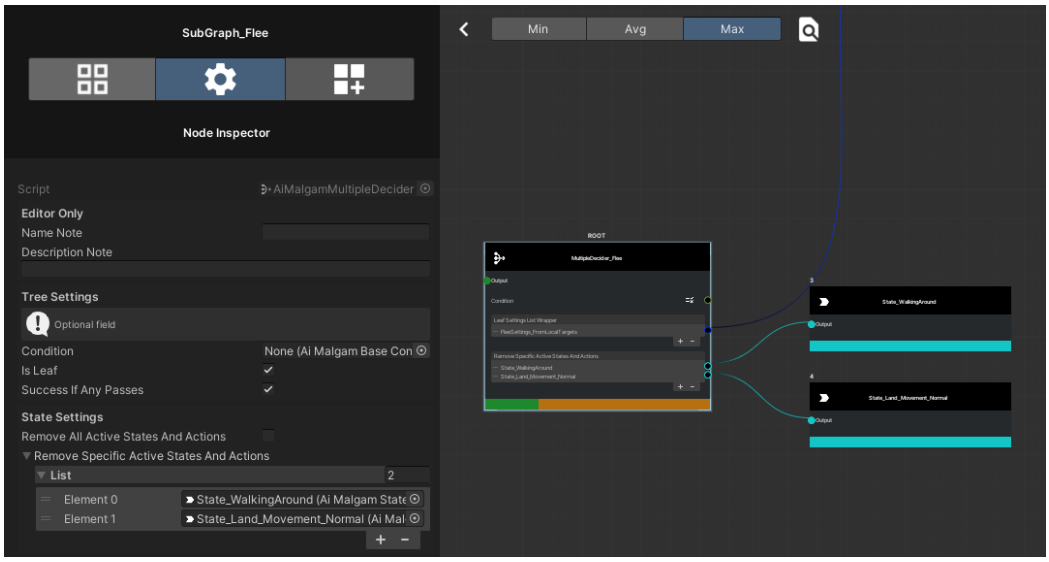
See the API here.

The usual Deciders either branch to other Decider nodes or Action settings, depending on if the "IsLeaf" field is checked or not:

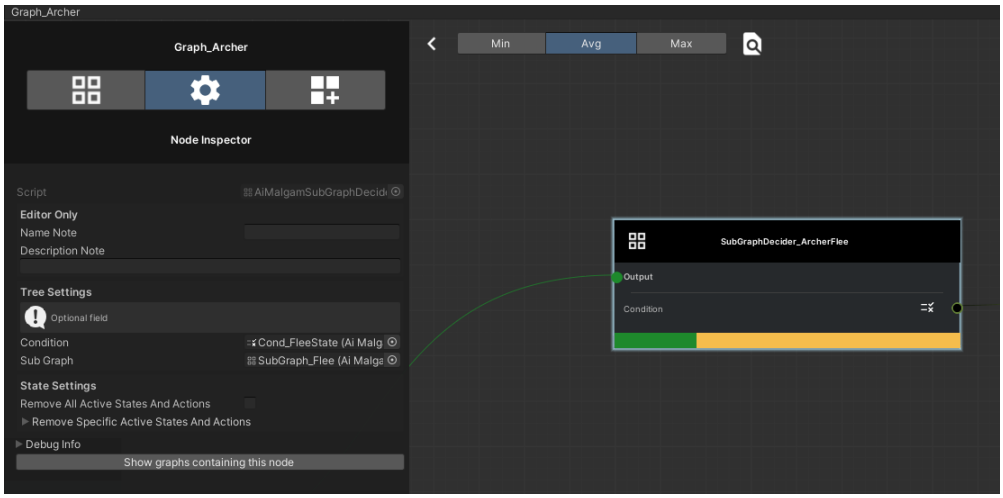




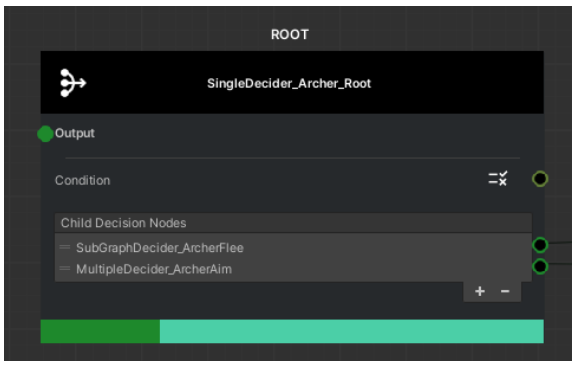
You can also remove States and Actions that are represented by those, if a Decider's traversal was successful:



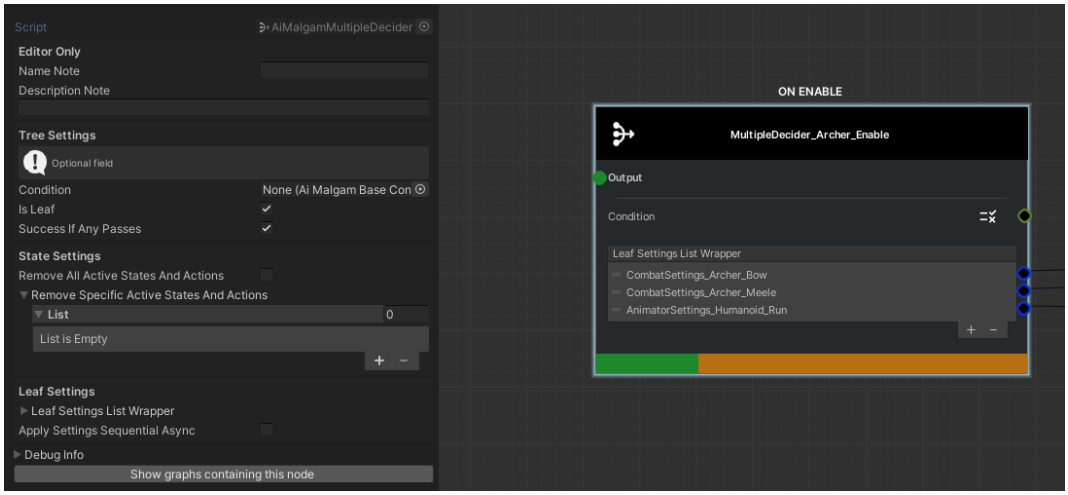
"Special" Deciders can also link to other graphs, like the Subgraph-Decider:



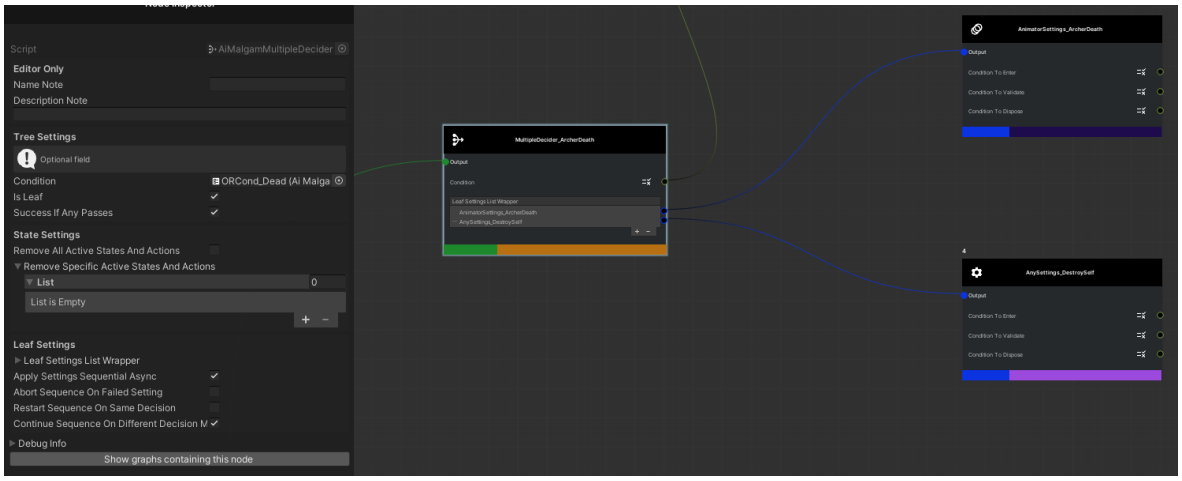
The **Single-Decider** only picks the first valid child (Decider or Action setting) and ignores the traversal of the other children in the list, just **like** the **selector task of a behavior tree**. Make sure to order your connections list correctly, since the order represents the traversal priority:



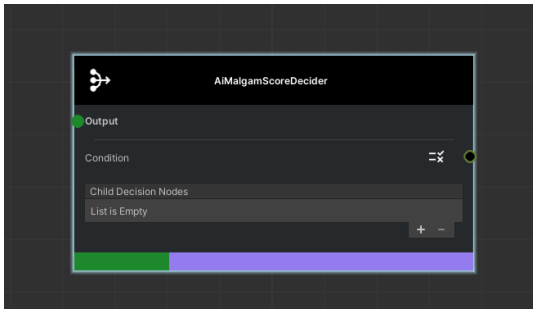
The **Multiple-Decider** node is **related to** the **sequence task of a behavior tree**. It by default tries to apply all children and invalidates if at least one child invalidated. You can turn that option off and validate, if at least one child was successful. The difference to the Single-Decider is that it will still apply the following child nodes in the list and not stop on the first successful:



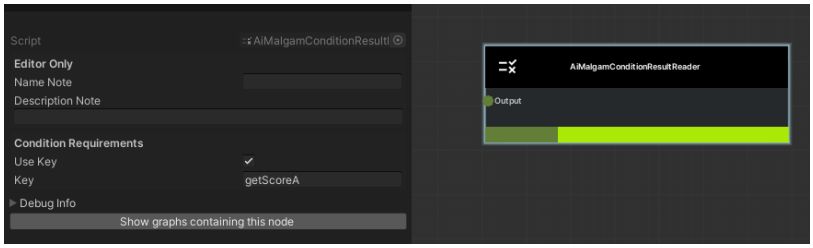
Additionally, you can apply the Action settings in an asynchronous sequence, where the upcoming Action in the list must wait on the previous Action to finish (see the previous **Section 2.5.2**):

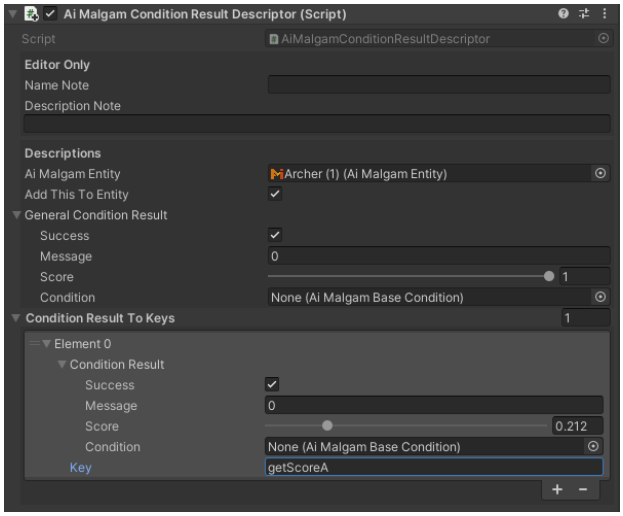


The **Score-Decider** reads the AiMalgamConditionResult score (see API) from either the Decider's Condition field or the "conditionToEnter" field found on Action settings to prioritize the decision traversal. This is a great way to implement own machine learning techniques that modify the Condition score outputs depending on if your AI was successful or could improve in some shape or form:

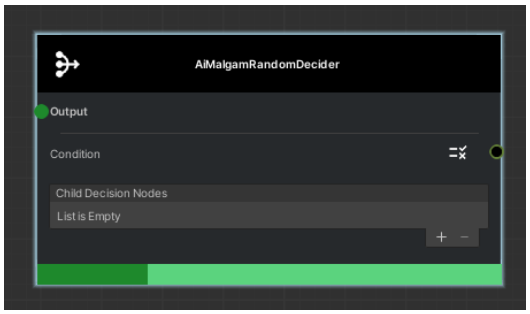


You can combine the Score-Decider with special Score-Conditions that read from the Score-Descriptor (MonoBehaviour) that is attached on the deciding AiMalgamEntity:





The **Random-Decider**, as the name suggests picks a child at random:



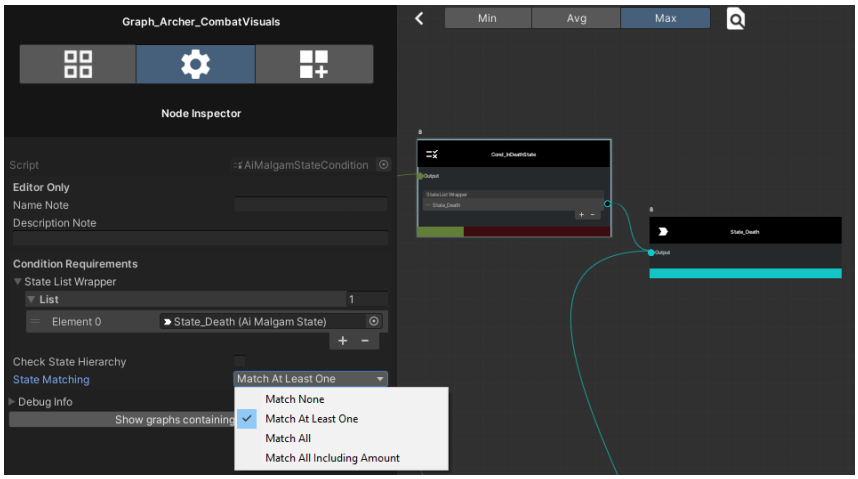
Condition Settings

See the APIs here and here.

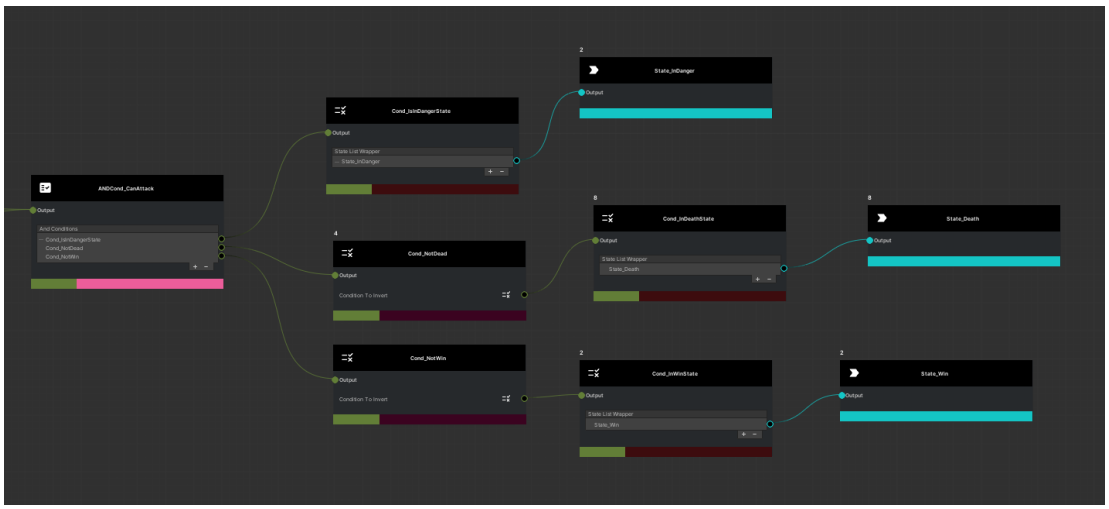
Condition settings offer a mechanism to validate or invalidate a certain given input in a modular and expandable manner by using them as an optional attachment for other (node) settings, reducing the number of nodes and layers in a tree, unlike how the conventional behavior tree pattern handles it.

The Condition setting's main method takes an AiMalgamEntity as its target to check the condition on and another entity that represents the calling Decider entity used for graph debugging.

Conditions usually store values that must match the provided values of the AiMalgamEntity runtime instance, for example checking if the required States are found on the target:

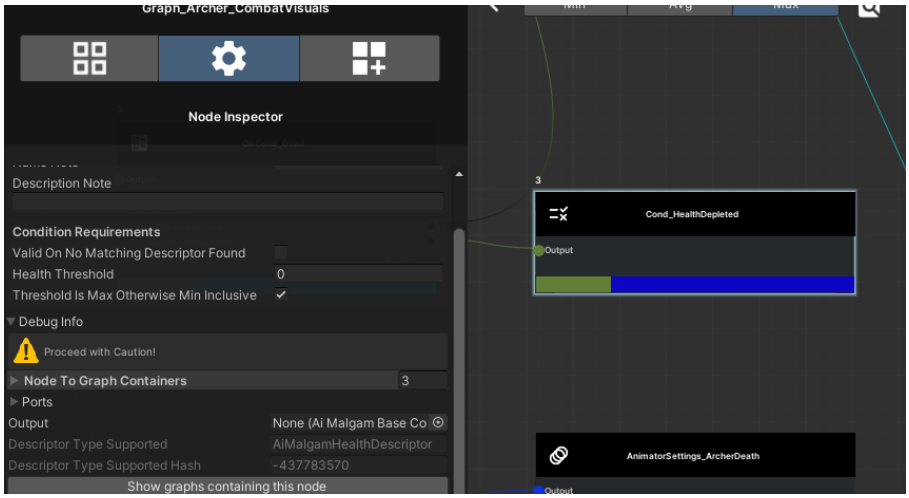


It is also possible to group Conditions by for example using the AiMalgamANDConditionContainer or the AiMalgamORConditionContainer (or creating a new grouping Condition type):

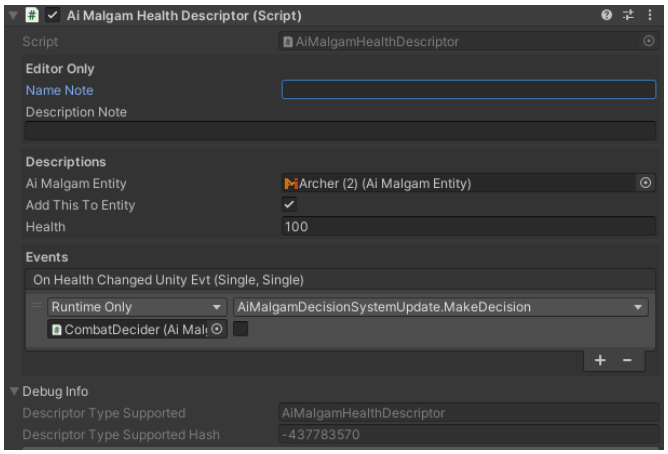


Both containers calculate the average scores of the checked child Condition results for the final Condition output.

Some Conditions require a Descriptor MonoBehaviour on an AiMalgamEntity to read special instance and runtime-based values, for example the health amount of an entity:



Such Conditions inherit from special Descriptor-Condition classes that automatically generate the Descriptor hash for you!



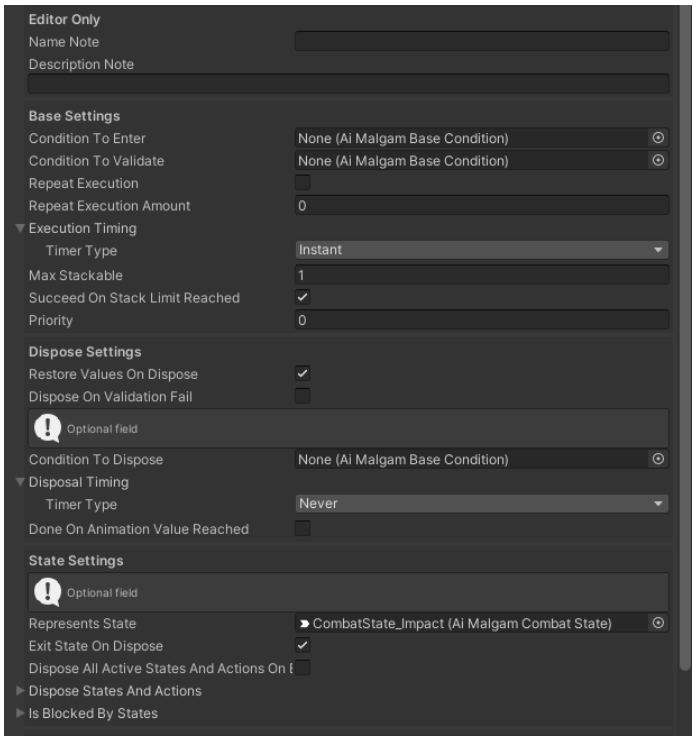
You can find the required Descriptor name and its generated hash in the Condition's "Debug Info" foldout section. The hash is then used as a parameter called in the "GetDescriptorByHash()" method found in the AiMalgamEntity class.

Action Settings

See the APIs here and here.

Action settings represent the leaf settings of the AI decision tree, containing information about the **execution frequency**, **life span**, and **input data** for the resulting Action that is managed by the scheduler (**Engine**).

Since each **Action** must be **maintained inside** an **Engine**, the same base Action setting applies for every custom Action:

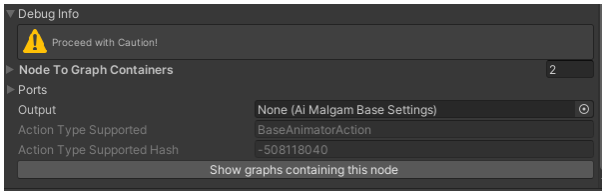


The base Action setting provides relevant life-cycle information about the resulting Action for the scheduler (**Engine MonoBehaviour**) and the Decision-System, such as:

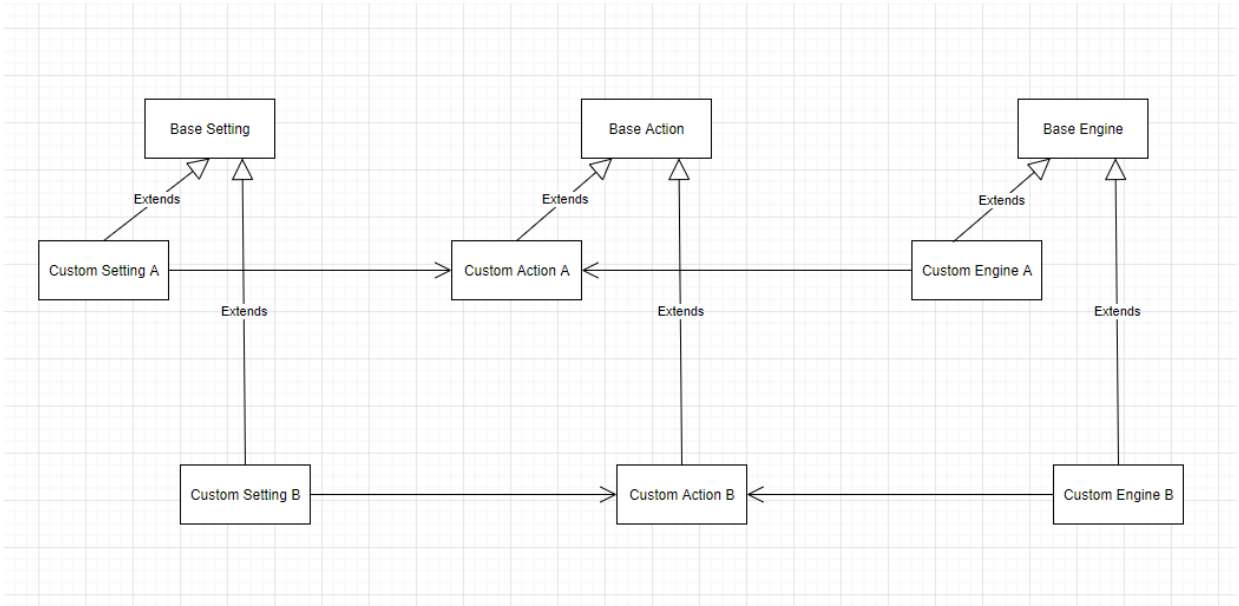
- The Condition to validate this setting before applying it to the Engine (upon entering)
- The Condition to validate this Action before each run cycle in the Engine (already applied)
- Infinite or limited repetition of this Action during its life cycle in the Engine
- Delay of execution, as well as the exact timings
- Max allowed Actions in one Engine at once sharing this exact setting (reference)
- Success or failure if the stack limit was reached
- The priority, indicating the execution order if multiple Actions are present within the same Engine
- Resetting any modified Components at runtime if the Action disposes and if there is anything to reset in the first place
- The disposal requirements indicating when and under which Conditions the Action should terminate
- The State setting represented by this Action (setting) to be able to find and remove it later on
- States and Actions, which should be removed upon entering the Engine
- States and Actions, which either prevent this Action from entering the Engine or terminate this Action when already running

Each Action setting defines its respective base Action class as a generic argument and generates a name and hash value from it automatically!
This is crucial to map the compatible Engine to this Action setting by calling the "GetEnginesByHash()" method found in the AiMalgamEntity class.

You can find this information in the setting's "Debug Info" foldout:



And here is the Action settings and Engines to Actions mapping diagram:

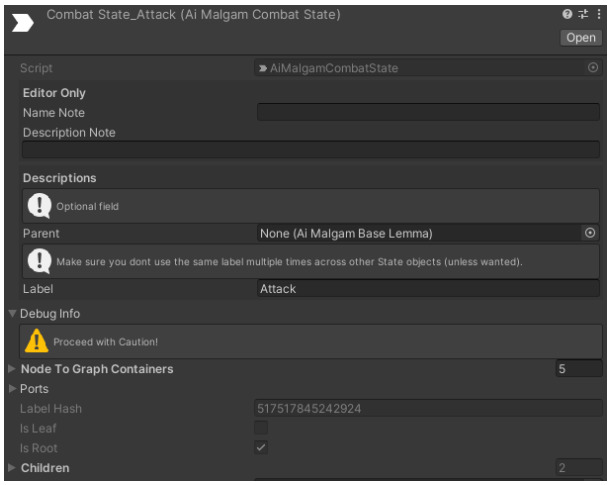


This video covers custom Action settings used in the demo that inherit from the AiMalgam core asset pack. You can create your own Actions with the Control Panel's code creation tools as explained in **Section 2.6.2**.

State Settings

See the API here.

The **State settings** include fields such as the label (string) for description, a labelHash (ulong) generated from the string for faster comparisons and an optional parent of type BaseLemma. The parent field enables to group State settings in a hierarchy for less maintenance and less logical checks if entire State groups can be found within one check.

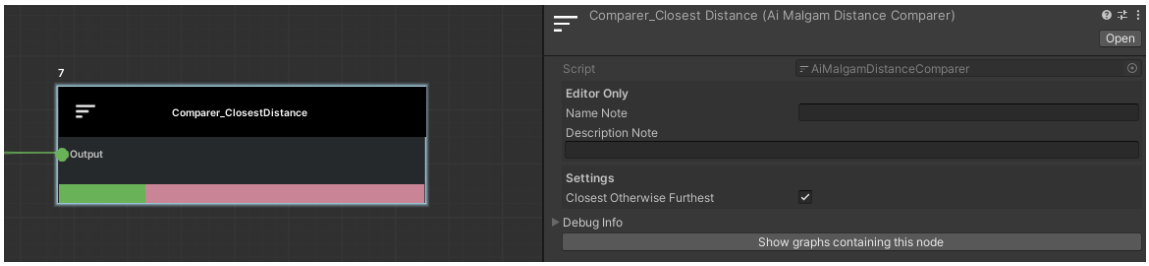


State settings are especially useful for Condition checks, knowing the running Actions of an AiMalgamEntity and finding the Actions to dispose them at any time!

Comparer Settings

See the APIs here and here.

The Comparer settings help you to sort or find an AiMalgamEntity by custom comparison checks, for example distance, States or names:



They are useful for follow or combat targeting, as it is implemented in the demo Follow-Action settings for instance.

Calling the “Compare()” method requires 3 parameters (see API). The first 2 being the list elements to compare and the 3rd one being a special entity to check the list item against (to determine the distance for example).

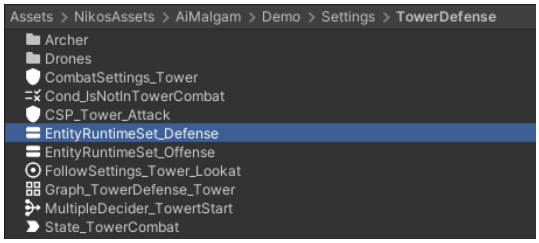
The “GetResult()” method will return the winning entity from the given list.

Runtime-Variable Settings

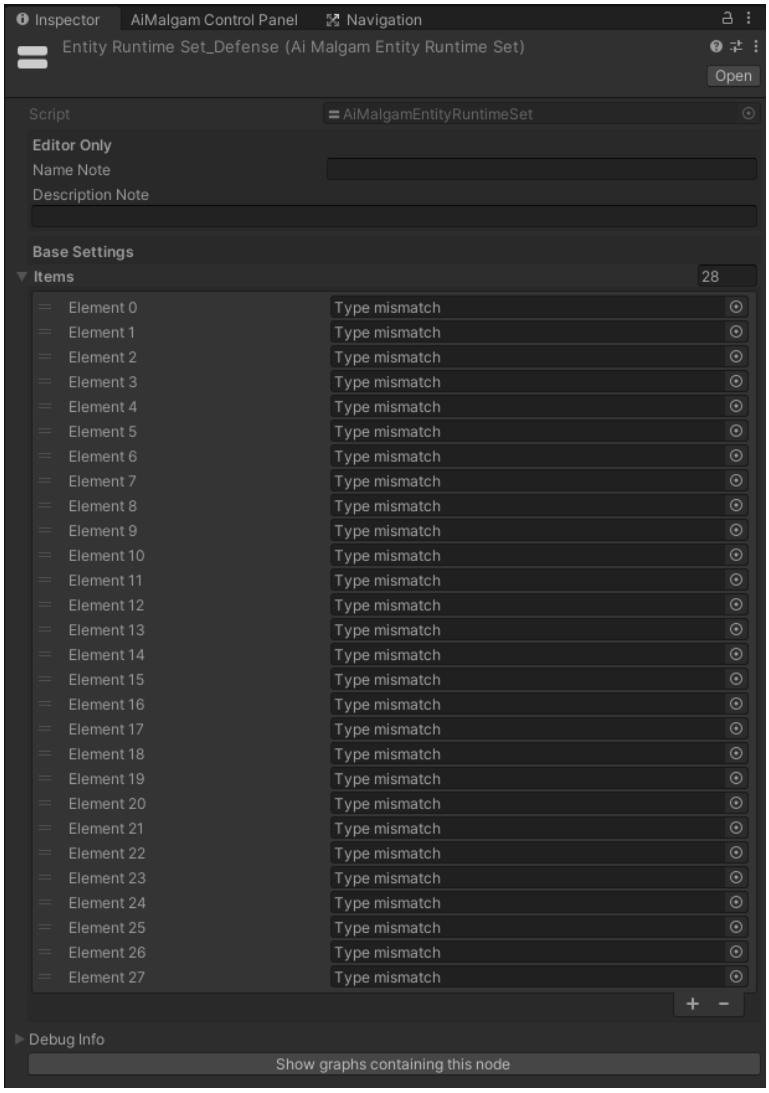
See the API here.

Runtime-Variables contain (primitive) data types or sets that can be shared across scenes and graphs to read global data that is modified at runtime. This approach is inspired by this GDC talk and covers an alternative architecture to for example singletons. It is used in the demo implementation but if you create your own custom AI settings, you are not forced or restricted to do so as well (you can for example read from Blackboards and Descriptors that reference singletons or whatever!!).

In the demo implementations they are mainly used to register and unregister dynamically added AiMalgamEntities in scenes to be considered in Action settings, such as following a list of dynamically added entities:



Unfortunately, GameObject references in ScriptableObject fields will always display the type mismatch name:

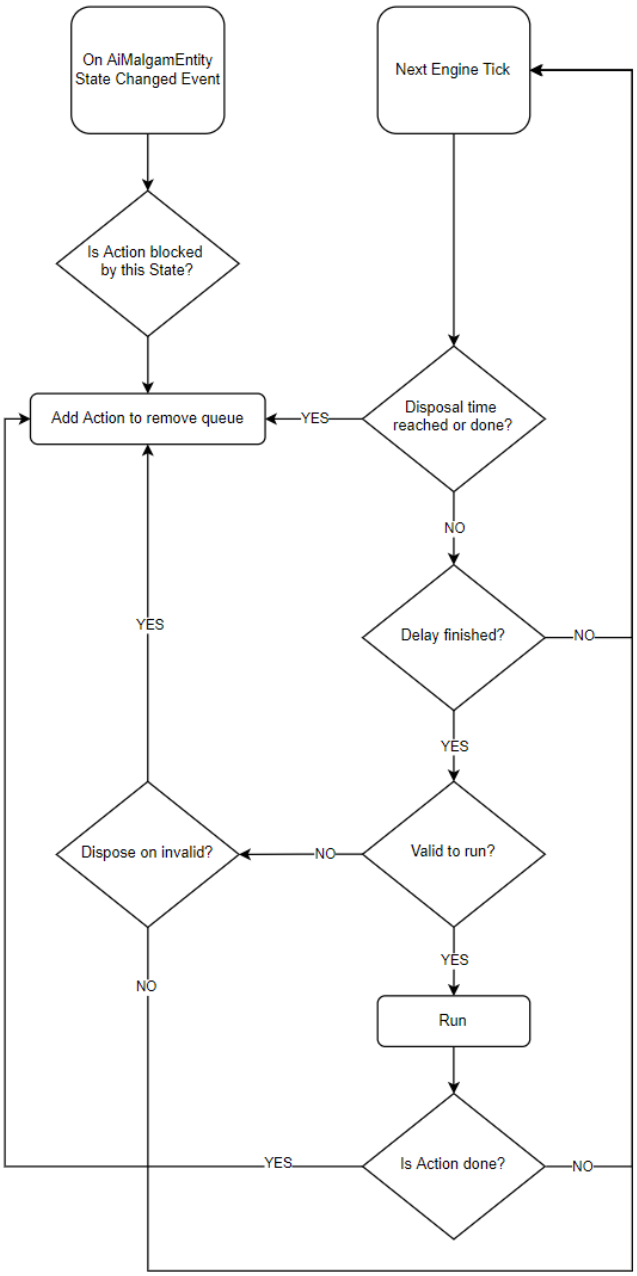


2.5.4 Actions

See the APIs here and here.

Action scripts are the custom AI logic implementation of their respective Action setting. An Action needs a Blackboard data type to read runtime and instance-based values from the respective Engine and entity (if any data is required in the first place).

The Life Cycle of an Action



2.5.5 Blackboards

See the APIs here and here.

When creating custom AI logic, or AI logic that is not known yet, a generic architecture is needed to offer the implementation and injection of custom data types and containers for that not yet known logic. The Blackboard pattern combined with the C# generic arguments is the perfect pattern for that!

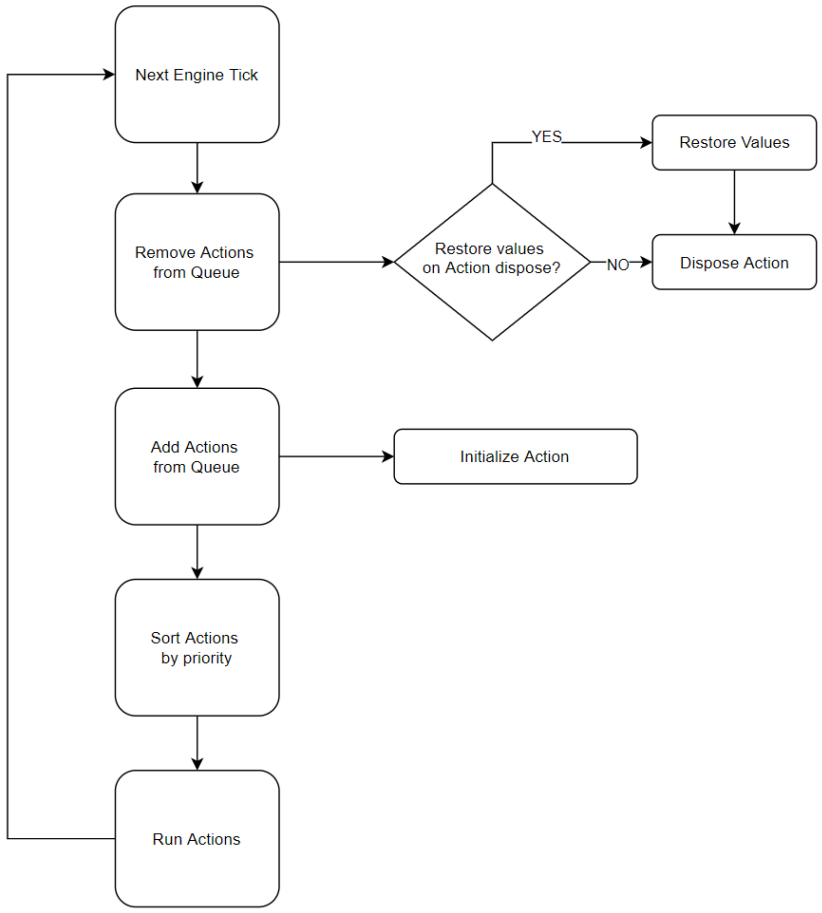
Blackboards are used to deliver runtime and instanced-based data retrieved from the Action's hosting Engine. This can for example be the current deltaTime, or special animation keys, or any other dynamically modified data value the Action needs.

2.5.6 Engines

See the API here.

Since this asset pack separates Action execution from decision-making (in this case tree traversal), a dedicated class is required that manages Actions after a decision was made.

The Tick Schedule of an Engine



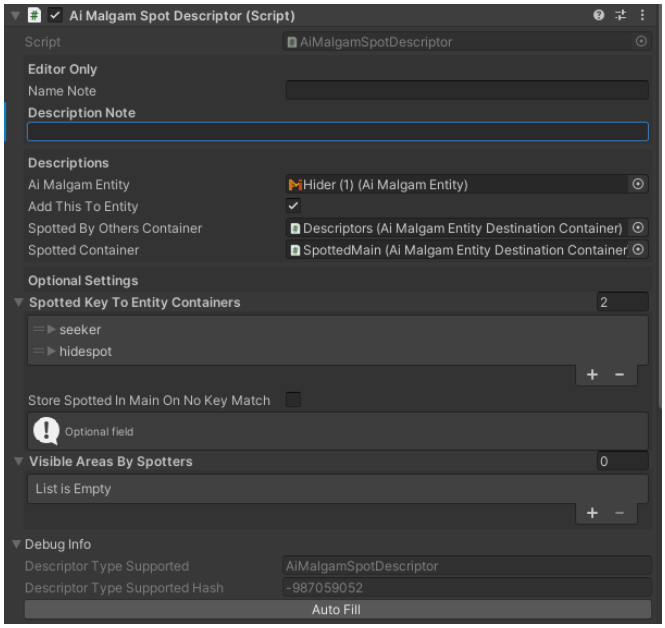
2.5.7 Descriptors

See the APIs here and here.

The Descriptor class is mainly used as a specific data container for Conditions but can also be used in combination with Engines, Actions or any other architecture.

Unlike the Blackboard implementation combined with Actions and Engines, the Descriptor class is an optional Unity Component and is usually not tied to any Engine or Action. Since the AiMalgamEntity only contains State settings to describe itself but not any other form of instance-based data, the Descriptor offers a (dynamic) extension to the AiMalgamEntity, enabling a modular approach to include custom runtime data.

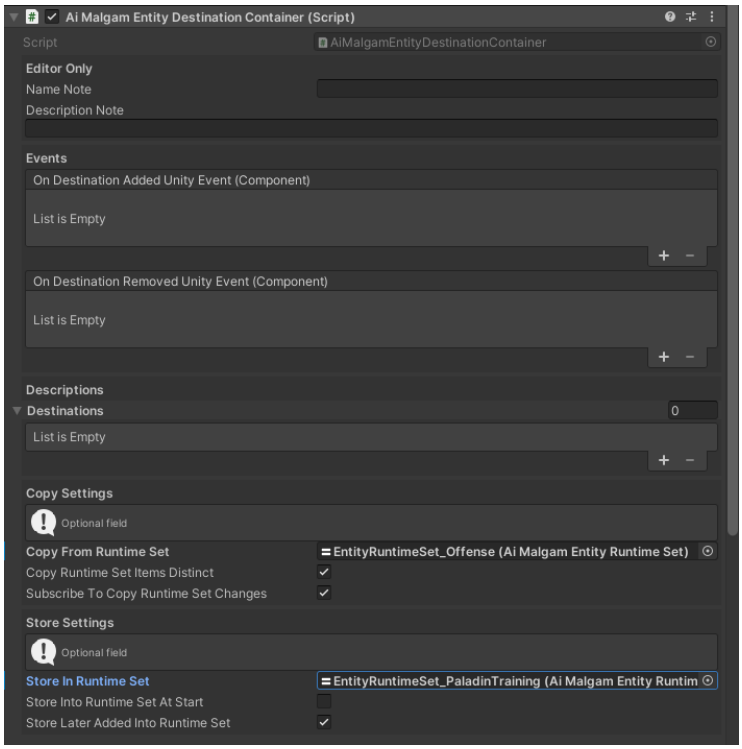
For example, containing information about spotted entities:



2.5.8 Additional Helper Classes

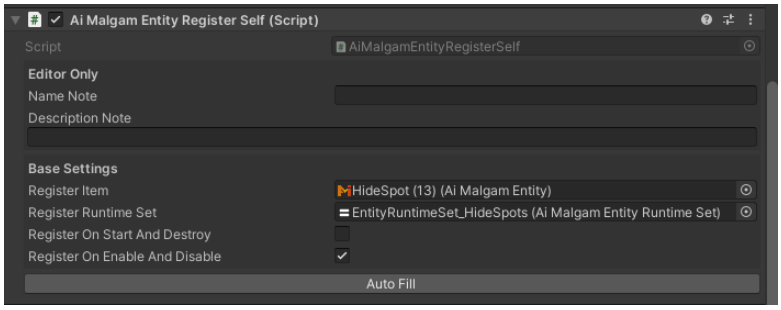
Destination Containers combined with Runtime-Sets (Node Settings)

Inheriting from the helpers Destination container (see API), you can create your own container that stores a list (distinct if wanted) of the given generic argument type and additionally emits events when the list was changed. This example stores AiMalgamEntities and furthermore enables you to read from Runtime-Sets (globally stored ScriptableObject AiMalgam node settings) either reacting to its changes or copying them initially into this container at runtime. Similarly, you can store your list items into a Runtime-Set as well:



Register Self to Runtime-Sets (Node Settings)

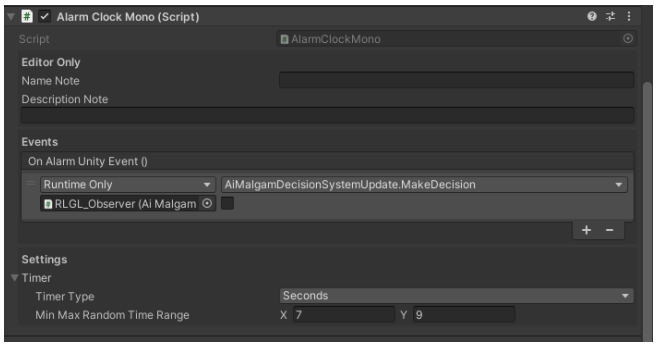
In some graphs you might want to use the Runtime-Set node settings to inject a list of for example AiMalgamEntities into an AiMalgam Action or Blackboard. This MonoBehaviour registers the given entity either at Start/ Destroyed or when Enabled/ Disabled:



This Component is used by many entities found in the Demo example scenes.

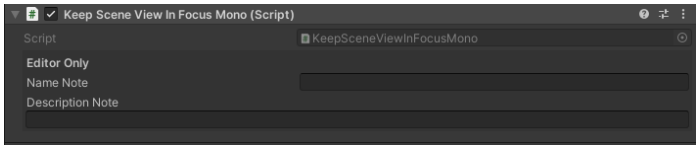
AlarmClock emitting Time Events

Found in the helper's dependency folder (see API), this MonoBehaviour allows you to setup a timed event, which can be used to for instance make a decision every x seconds or minutes:



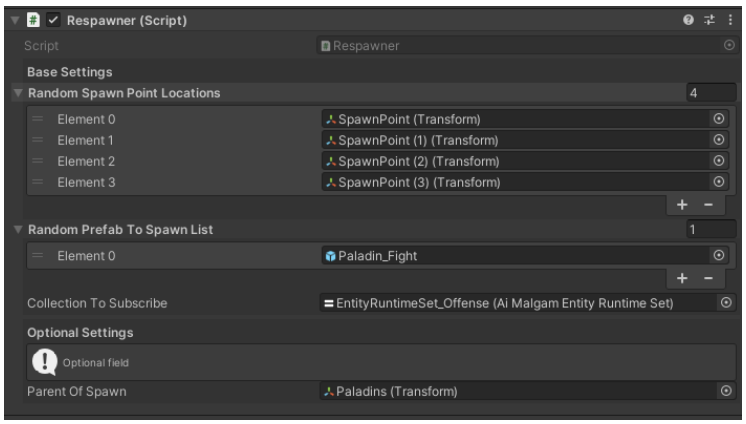
Keep the Scene View focused when entering Play Mode

Also found in the helper's dependency (see API), this script when enabled prohibits the Unity Editor to focus the Game window:



Respawn removed Entities

In some demo scenes removed AiMalgamEntities from a Runtime-Set node setting will respawn another entity prefab at some random pre-defined spawn points (see API):



2.6 Working with the Control Panel and generating custom AI scripts

It is recommended to watch the video tutorial series additionally to this manual.

With the **AiMalgam Control Panel** you can **view**, **locate** and **delete** every **graph and node setting** stored in your project. Additionally, you can create your own AI logic via the code creator tab.

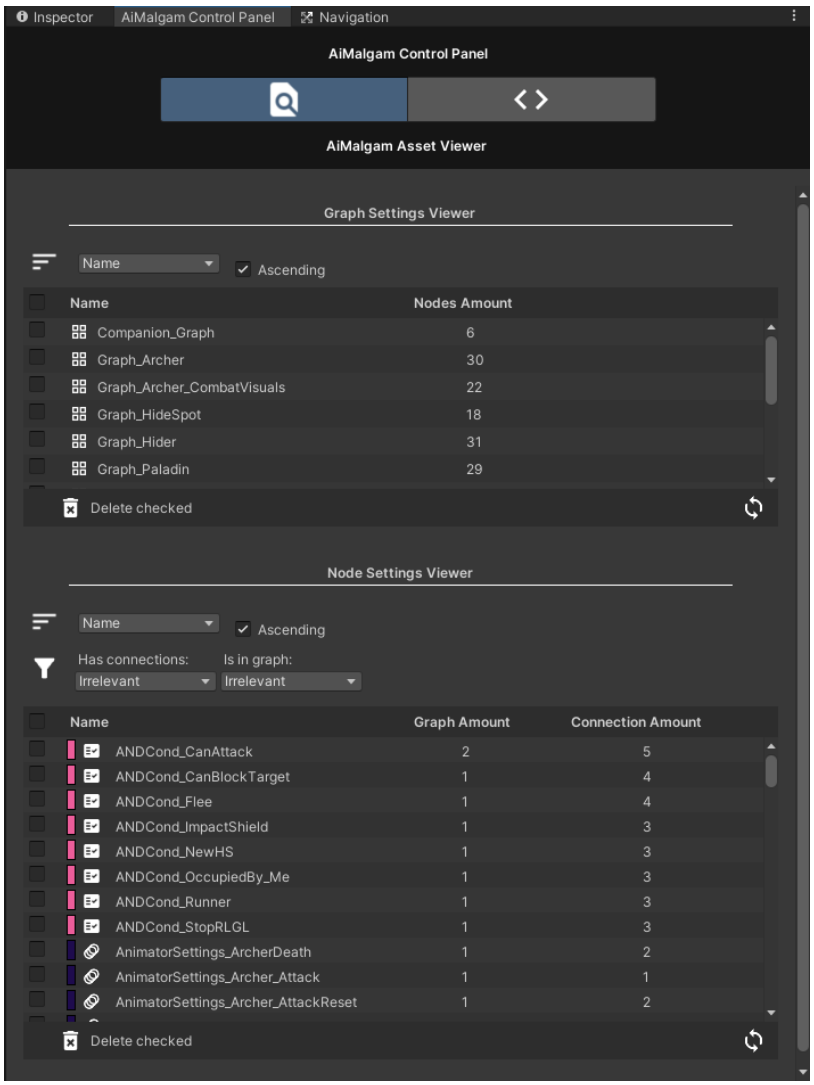
2.6.1 AiMalgam Asset Viewer

In the Asset Viewer tab, you have 2 separate lists, one containing graph assets and the other node setting assets. Make sure to check the refresh icon at the bottom right corner of the respective list to update newly created AiMalgam assets!

Selecting an item in one or the other list, will select it in your project and locate it as well.

You also can sort both lists by their offered dropdown options and special column information. Additionally, the node list offers you to filter certain nodes by their connections amount and in graphs amount if you like.

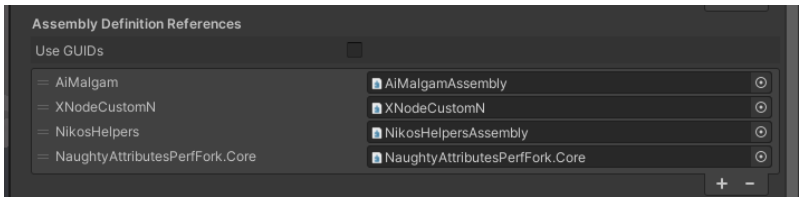
Deleting assets works by first checking the checkbox of the assets you wish to delete and finally by pressing the delete checked button at the bottom of the respective list. A new **popup window** will appear, informing you that this **action cannot be undone** and that the **undo/redo history** will be **cleared** as well. Also note that the **ScriptableObjects** you want to delete **might be referenced** in non-connection setups, like in MonoBehaviours or other ScriptableObjects (Only if you have set this up)!



2.6.2 AiMalgam Code Creator

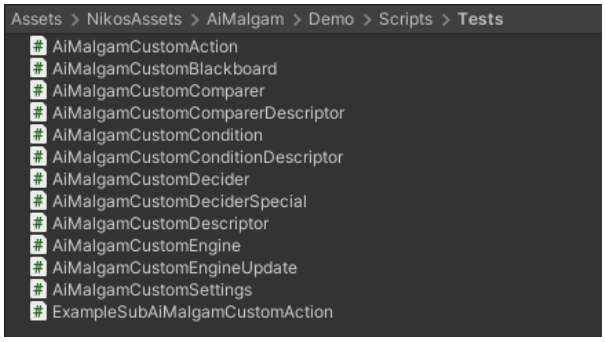
It is highly recommended to watch the video series for better code creation understanding!

If you are using assembly definitions, make sure to reference those assembly definitions where you output the generated code:



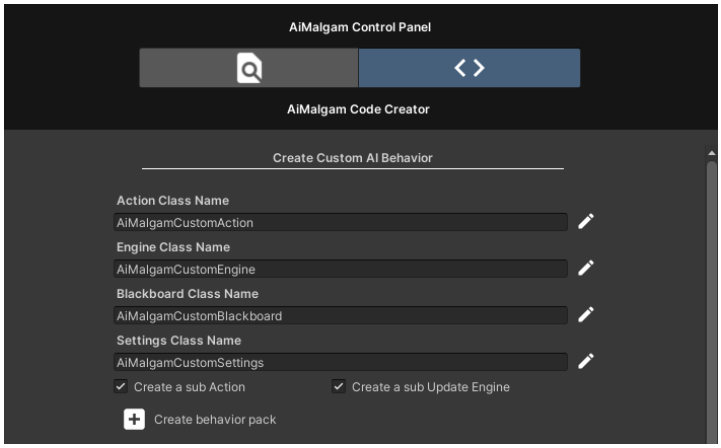
Before you can generate a new script file, you first need to define a **class and file name** (**input field**) and set the **file's location** within your project (**edit icon** to the right). Finally, press the **create button** (rectangular "+" icon at the bottom of each section) **to generate the script(s)**.

All scripts generated in the same folder example:

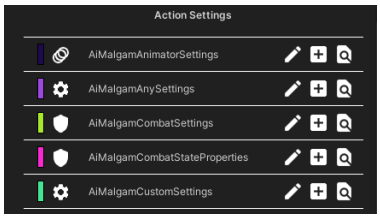


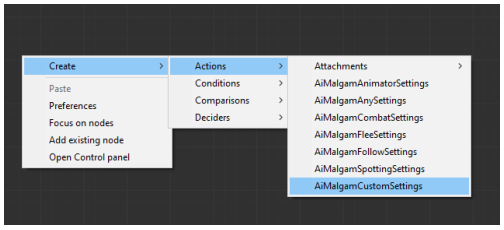
Creating new AI Behavior Packs

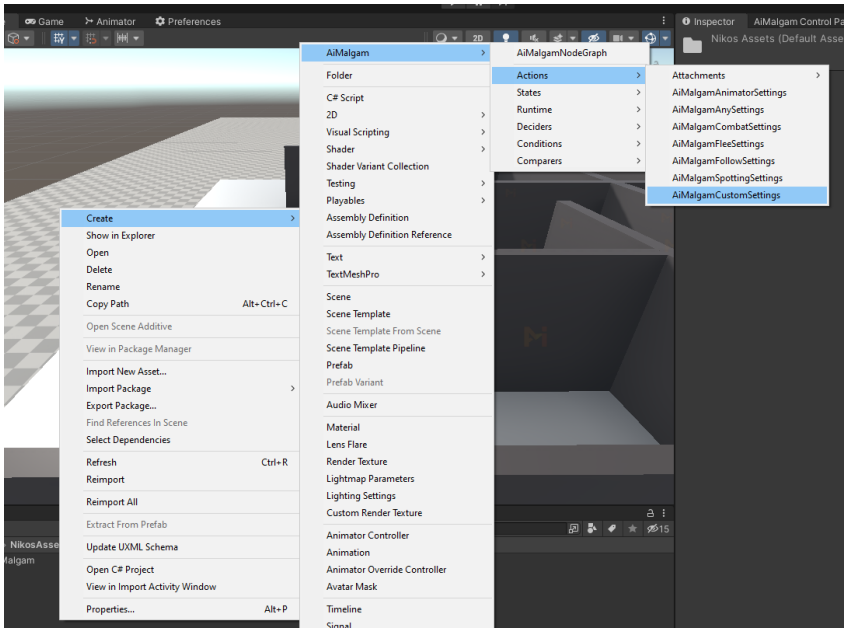
To create new AI logic, use the following displayed section. The 2 checkboxes below create specific (non-abstract) Action and Engine implementations that inherit from their respective base classes.

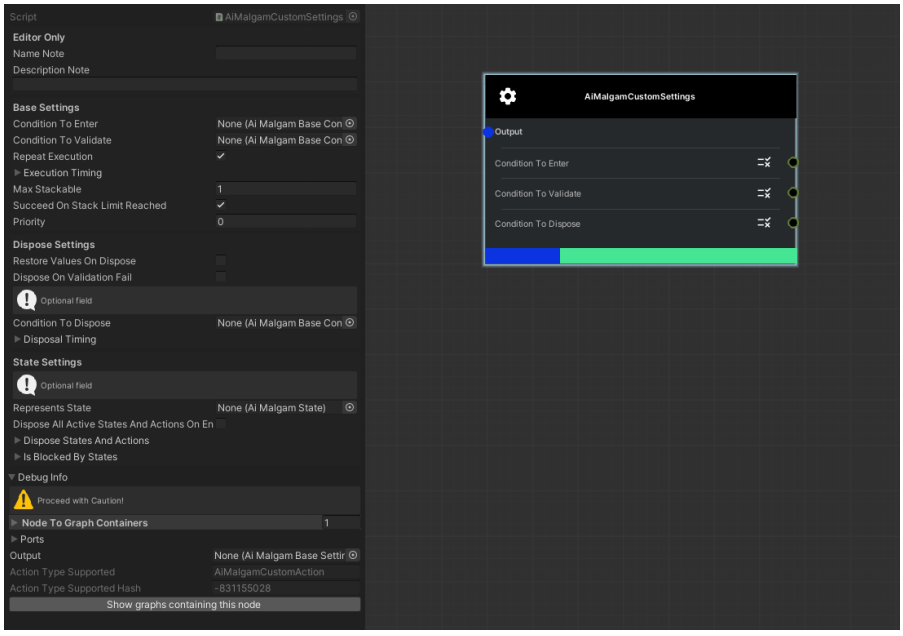


After pressing the “Create behavior pack” button, you can add your custom Actions via the graph’s and project’s context menu, as well as the graph’s side panel:

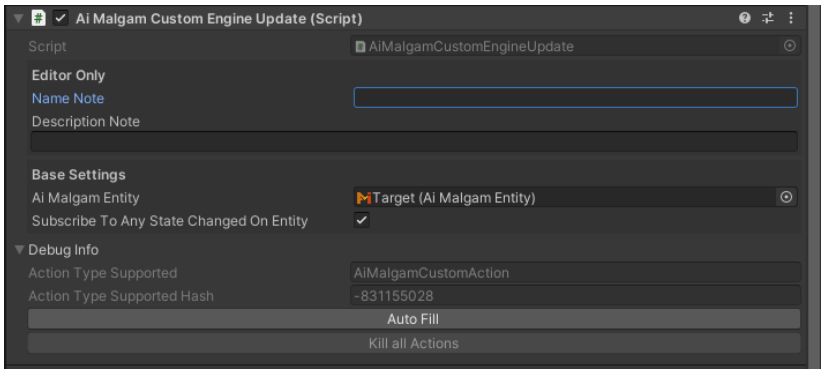








The “sub-Update” Engine (checkbox = true) can be added as a MonoBehaviour script to your GameObject:

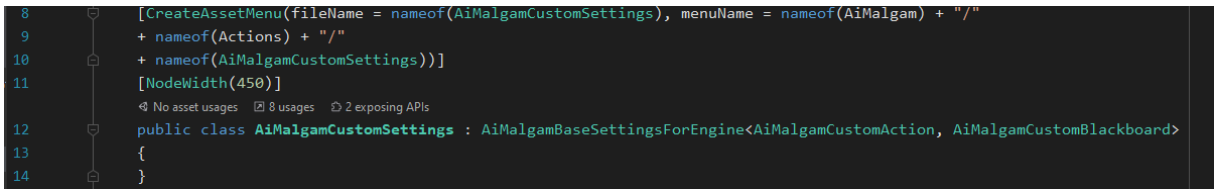


Generated Action Node Settings

As you can see in the following illustration, the custom generated Action settings inherit the AiMalgamBaseSettingsForEngine class (line 12) and define the required generic arguments as the also and alongside generated custom Action and Blackboard types. This is very important to establish the hash value later used by the AiMalgamEntity to map the Action setting to the respective Engine without using reflection calls whilst mapping!

In the class body you can add all sorts of settings for your custom Action!
The **class** and **field attributes** will be explained later in **Section 2.6.3**, showing you how to offer **custom input ports** and input **port lists**.

For generated settings you do **not** need to define an **extra output field**! This is only relevant if you want to further specify your setting or create a blank node that inherits directly from AiMalgamNode.



```
8       [CreateAssetMenu(fileName = nameof(AiMalgamCustomSettings), menuName = nameof(AiMalgam) + "/"
9       + nameof(Actions) + "/"
10      + nameof(AiMalgamCustomSettings))]
11      [NodeWidth(450)]
        No asset usages  8 usages  2 exposing APIs
12      public class AiMalgamCustomSettings : AiMalgamBaseSettingsForEngine<AiMalgamCustomAction, AiMalgamCustomBlackboard>
13      {
14      }
```

Generated Custom Base Action

The custom Action allows you to interact with the game environment on an instance-based level at runtime. See the **life cycle diagrams** in the **Sections 2.5.4** and **2.5.6** to understand when the methods are called.

The **Initialize method** (line 59) is called only once the Action is applied to the respective Engine.

The **Restore method** (line 66) is called right before the Actions disposal and only if it was checked in the represented Settings.

The **Validate method** (line 71) is called right before each run method call and can lead to skipping the Run method, if not even disposing the Action if it invalidates/ returns false (depending on the Actions settings setup). Note that the Condition to validate setting is called in the Engine separately, so you don't have to implement it in each validate method.

```
8   public abstract class AiMalgamCustomAction : AiMalgamBaseAction<AiMalgamCustomBlackboard>
9   {
10      public override event Action<IAiMalgamBaseActionWrapper> OnActionInitialized;
11
12      Properties
53
          1 usage
54      public AiMalgamCustomAction (AiMalgamCustomSettings customSettings, AiMalgamCustomEngine customEngine) : base(customSettings, customEngine)
55      {
56
57      }
58
          Frequently called  0+1 usages
59      public override void Initialize(AiMalgamCustomBlackboard blackboard)
60      {
61          //Optional Initialization
62
63          this.OnActionInitialized?.Invoke(obj:this);
64      }
65
          Frequently called  0+1 usages
66      public override void Restore(AiMalgamCustomBlackboard blackboard)
67      {
68          //Optional Restoration
69      }
70
          Frequently called  0+1 usages
71      public override bool Validate(AiMalgamCustomBlackboard blackboard)
72      {
73          //Optional Validation
74
75          return true;
76      }
77  }
```

Generated Custom "sub-Action"

Here you see the generated "sub-Action" (checkbox = true) that includes the **Run method** (line 15) where you would execute your desired AI entity logic!

The **Success property** as shown in line 20 is relevant for the async life cycle as shown in **Section 2.5.2**, where async sequential Actions wait for the previous Action to dispose (successfully) as defined in the Multiple-Decider node setting.

```
8   public class ExampleSubAiMalgamCustomAction : AiMalgamCustomAction
9   {
10      public ExampleSubAiMalgamCustomAction(AiMalgamCustomSettings customSettings, AiMalgamCustomEngine customEngine)
11          : base (customSettings, customEngine)
12      {
13      }
14
          Frequently called  0+1 usages
15      public override void Run(AiMalgamCustomBlackboard blackboard)
16      {
17          //do your sub actions specific work
18
19          //was your action successful?
20          Success = true;
21      }
22  }
23  }
```

If you want to interrupt the Actions life cylce, you can simply set the following Action property to true. In the next Engine Tick the Action will then be disposed, as you can see in the **life cycle** diagrams in **Sections 2.5.4** and **2.5.6**.

```
this.IsDone = true;
```

Generated Custom Blackboard

Blackboards are used to transfer data prepared from the Engine to its running Actions, as shown in the previous Action object methods. The Actions can also write data to the Blackboard to make it accessible for other Actions or other Scripts that read from the Engines Blackboard property.

The generated Blackboard is by default empty since you don't need to put data in there unless required by your Action logic.

```
3       public class AiMalgamCustomBlackboard : IAiMalgamBlackboard
4       {
5           //Optional Values For Actions To Read From
6       }
```

Here is a Blackboard example for the demo movement Actions (flee and follow), setting some primitive data and Component references as well:

```
7       /// <summary>
8       /// A wrapper <see cref="IAiMalgamBlackboard"/> containing data for various movement mechanics
9       /// </summary>
            9 usages  2 inheritors  0+4 exposing APIs
10      public class MovementBlackboard : IAiMalgamBlackboard
11      {
12          public Transform transform;
13          public NavMeshAgent navMeshAgent;
14          public Rigidbody rigidbody;
15
16          public float deltaTime = 1;
17          public float displayFXTimeInSec = .5f;
18          public bool shouldDebugInEditor;
19
20          public BlendingHelper blending;
21          public bool clampHeight;
22          public Vector2 minMaxHeightThreshold = new Vector2( x: 5,  y: 10);
23          public ForceMode forceMode;
24      }
25  }
```

Generated Custom Base Engine

The generated Engine (MonoBehaviour) is needed to host and manage your custom Actions. It extends the AiMalgamEntityBaseEngine class and sets your custom generated Action and Blackboard classes as the generic arguments (line 8) to calculate the hash value later used in mapping the custom Action setting to this Engine.

The generated Blackboard type can then be accessed and modified with the overridden property at line 12.

```
8  public abstract class AiMalgamCustomEngine : AiMalgamEntityBaseEngine<AiMalgamCustomAction, AiMalgamCustomBlackboard>
9  {
10     //Optional Engine and Action Specific Components
11
       1 usage
12     public override AiMalgamCustomBlackboard Blackboard
13     {
           Frequently called
14         get;
15         set;
16
17     } = new AiMalgamCustomBlackboard();
18
```

To convert your custom Action settings to your custom Action, you need to modify the "ApplySettings()" method featured in line 19. Usually, you just want to create a new instance of your custom Action, either always the same or as shown in the illustration via a switch statement that reads an enum (or any other data) from your custom settings. This is also done in the demo spotting and follow Engines.

Line 21 and 40 define a result object used in the leaf traversal mechanic and are very important for the decision-making process (returned in line 48). Make sure that this success or failure validation matches your custom implementation. Usually, this implementation is the way to go.

In line 44 we must add the instantiated Action to this Engines queue to later be run in the "Tick()" method.

Line 46 marks the given custom Action node settings in the graph, as long as the selected Decision-System Component hosts the same opened graph editor (see **Section 2.4**). Theoretically this is optional but why should you miss out debugging this Action setting node in the graph editor?

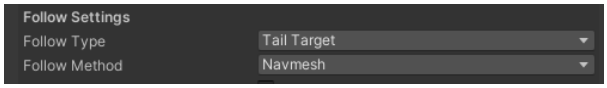
```
19     public override AiMalgamApplyActionSettingResult ApplySettings(AiMalgamBaseSettings settings)
20     {
21         AiMalgamApplyActionSettingResult applySettingResult = new AiMalgamApplyActionSettingResult(settings, success: true);
22         AiMalgamCustomSettings customSettings = (AiMalgamCustomSettings)settings;
23         //the next action to be added into the agentai engine
24         AiMalgamCustomAction customAction = null;
25         /* Example to choose the right action
26         switch (//customSettings.YourActionsDeciderEnum)
27         {
28             case AiMalgamCustomSettings.YourActionsDeciderEnum.USE_X:
29                 customAction = new ExampleSubAiMalgamCustomAction(customSettings, this);
30                 break;
31             case AiMalgamCustomSettings.YourActionsDeciderEnum.USE_Y:
32                 customAction = new AiMalgamCustomAction_Y(customSettings, this);
33                 break;
34             case AiMalgamCustomSettings.YourActionsDeciderEnum.USE_Z:
35                 customAction = new AiMalgamCustomAction_Z(customSettings, this);
36                 break;
37         }*/
38
39         applySettingResult.actionResult = customAction;
40         applySettingResult.success = customAction != null;
41         applySettingResult.message = AiMalgamApplyActionSettingResult.MESSAGE_CHOOSE_ACTION;
42
43         if (applySettingResult.success)
44             this.AddActionsQueue(CollectionHelper.ListOfOne(customAction));
45
46         settings.MarkInGraph(this.AiMalgamEntity, applySettingResult.success);
47
48         return applySettingResult;
49     }
```

Generated Custom "sub-Engine"

If you checked the box to generate a sub-Engine, this is the generated class. It only consists of an "Update()" method that calls the base Engine's "Tick()" method. You can handle the Engine's life cycle at any time you want, it doesn't have to be the Unity3D default "Update()" method!

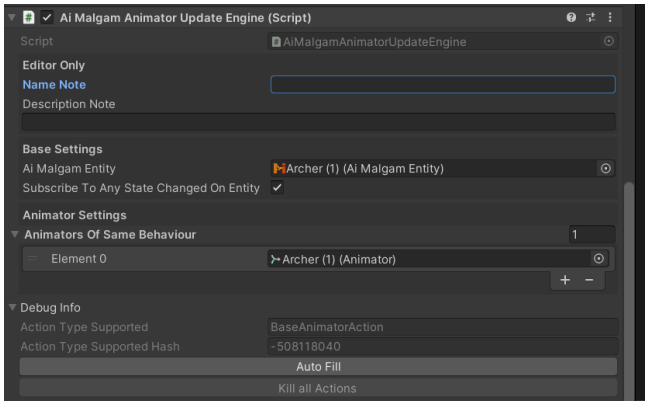
```
public class AiMalgamCustomEngineUpdate : AiMalgamCustomEngine
{
    Event function
    private void Update()
    {
        //Do stuff with the Blackboard if needed...

        this.Tick(this.Blackboard);
    }
}
```

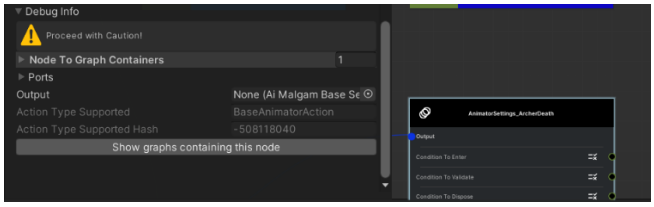
Here is an example Engine handling the different follow Actions from one follow Action setting:




```
switch (followSettings.followType)
{
    case AiMalgamFollowSettings.FollowType.TailTarget:
        followAction = new FollowTailAction(followSettings, followEngine: this);
        break;
    case AiMalgamFollowSettings.FollowType.RandomPoint:
        followAction = new FollowRandomAction(followSettings, followEngine: this);
        break;
    case AiMalgamFollowSettings.FollowType.JustLookAtTarget:
        followAction = new JustLookAtFollowAction(followSettings, followEngine: this);
        break;
    case AiMalgamFollowSettings.FollowType.SimpleFlankTarget:
        followAction = new SimpleFlankTargetAction(followSettings, followEngine: this);
        break;
}

applySettingResult.actionResult = followAction;
applySettingResult.success = followAction != null;
applySettingResult.message = AiMalgamApplyActionSettingResult.MESSAGE_CHOOSE_ACTION;

if (applySettingResult.success)
    this.AddActionsQueue(CollectionHelper.ListOfOne(followAction));
```

If you unfold the "Debug Info" foldout found in the generated Action setting node and the generated Engine MonoBehaviour, you will find the hash value used for mapping the generated Action setting to its Engine via the AiMalgamEntity:

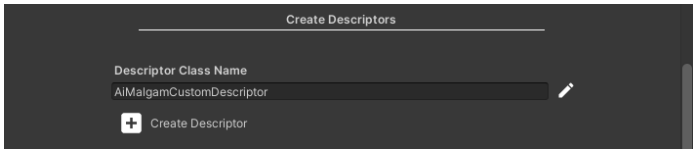




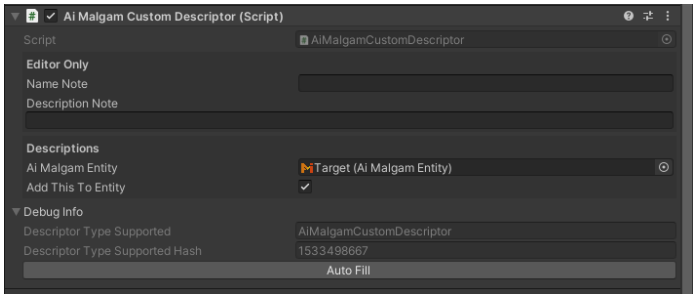
Creating new Descriptors

The Descriptor class is your custom dynamic data container for the AiMalgamEntity instance since by default the AiMalgamEntity only represents its current AiMalgamStates with just their label information. Adding a Descriptor can help you define a more specific entity and use it in Conditions, therefore in the decision-making process as well.

Like in the previous script generation section, you need to define a path and the file name. Make sure to reference the assembly definition files if you use one yourself!



You can add the generated script to any GameObject. Make sure to set the hosting or extending AiMalgamEntity either via the “AutoFill” button found in the Inspector or by assigning the field on your own.



The generated script is empty by default and only inherits from the base Descriptor class in line 3:

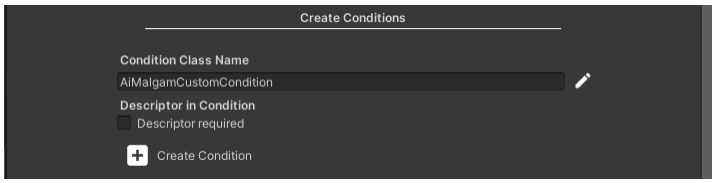
```
3   public class AiMalgamCustomDescriptor : AiMalgamBaseDescriptor
4   {
5       //Add Your AiMalgam Entity Specific Values To Describe Its State
6   }
```

Here is an example showcasing a Health-Descriptor used in the demo samples to keep entities alive or kill them off via their respective graphs:

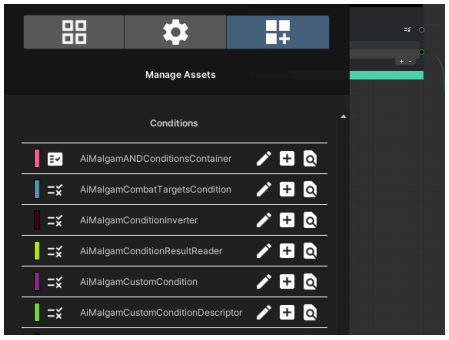
```
15  /// <summary>
16  /// A specific <see cref="AiMalgamBaseDescriptor"/> to display health of an <see cref="AiMalgamEntity"/>
17  /// </summary>
    4 asset usages   8 usages
18  public class AiMalgamHealthDescriptor : AiMalgamBaseDescriptor
19  {
20      public event Action<float, float> OnHealthChanged;
21      [BoxGroup( name:AiMalgamConstants.ATTRIBUTE_FIELD_BOXGROUP_EVENTS)]
22      public OnHealthChangedUnityEvt OnHealthChangedUnityEvt = new OnHealthChangedUnityEvt();  3 methods
23
24      [SerializeField]
25      [BoxGroup( name:AiMalgamConstants.ATTRIBUTE_FIELD_BOXGROUP_DESCRIPTIONS)]
26      protected float _health = 100;  Changed in 3+ assets
27
        2 usages
28      public float Health
29      {
            Frequently called
30          get => _health;
31          set
32          {
33              float prevHealthVal = _health;
34
35              _health = value;
36
37              //we only want to call the event after the health has been changed to avoid wrong Health readings
38              OnHealthChanged?.Invoke(value, prevHealthVal);
39              OnHealthChangedUnityEvt?.Invoke(value, prevHealthVal);
40          }
41      }
```

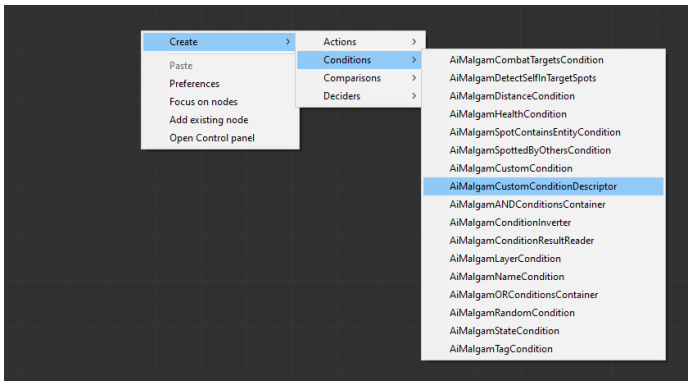
Creating new independent Condition Node Settings

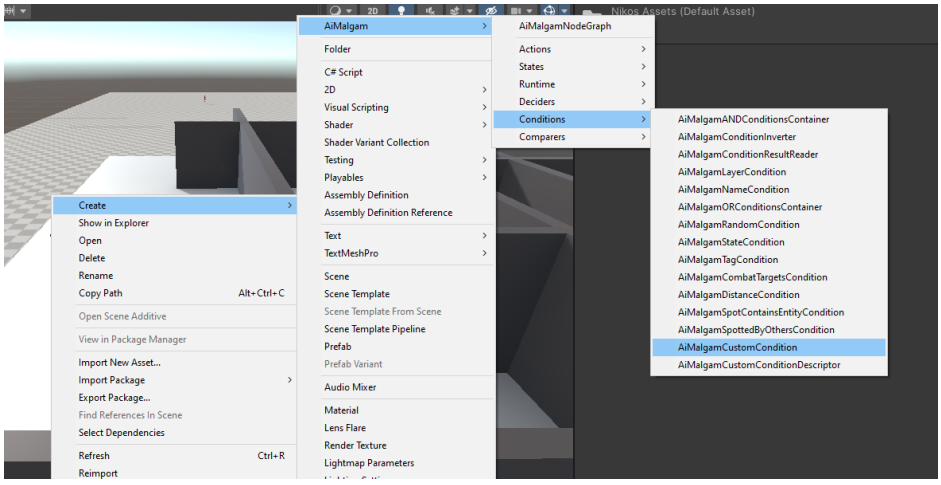
You can either create independent Condition node settings or Descriptor dependent ones:



The generated Condition settings can be added via the context menus or via the manage asset side panel found in the graph editor:







Every AiMalgam Condition node setting offers the **"IsConditionMetFor()"** method (as seen in line 13 and in the API) which takes 2 AiMalgamEntity objects as parameters, one for the actual condition check and the other to allow or disallow marking this Condition setting in an opened graph editor while in the Unity Editor and in play-mode. During the tree traversal, both given entities are the same. Only in special circumstances like in follow or spotting Actions are those 2 entities different.

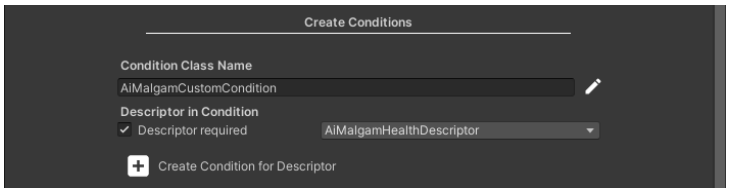
Add your custom values to check in the class body and do the check logic in the method:

```
6   [CreateAssetMenu(fileName = nameof(AiMalgamCustomCondition), menuName = nameof(AiMalgam) + "/"
7       + nameof(Conditions) + "/"
8       + nameof(AiMalgamCustomCondition))]
        No asset usages   2 usages
9   public class AiMalgamCustomCondition : AiMalgamBaseCondition
10  {
11      //Add Your Check Against Values Here
12
            Frequently called   0+33 usages
13      public override AiMalgamConditionResult IsConditionMetFor(AiMalgamEntity targetAiMalgamEntity,
14          AiMalgamEntity conditionsRequesterEntity)
15      {
16          AiMalgamConditionResult conditionResult = base.IsConditionMetFor(targetAiMalgamEntity, conditionsRequesterEntity);
17
18          //Add your special condition here
19
20          //Mark success or failure in graph
21          this.MarkInGraph(conditionsRequesterEntity, conditionResult.success);
22          return conditionResult;
23      }
24  }
25  }
```

Creating new Descriptor dependent Condition Node Settings

To create a Descriptor dependent Condition node setting, tick the checkbox below and select your desired Descriptor that must be present on the target AiMalgamEntity to read its data and validate the Condition.

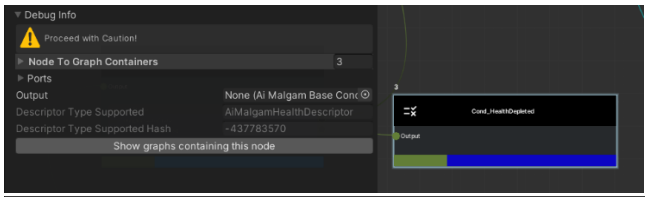
If you use Descriptors from the demo implementation, make sure to reference the demo assembly definition file to yours, as long as the resulting path is within your own assembly definition scope. If you don't use any assembly definitions, just ignore this.

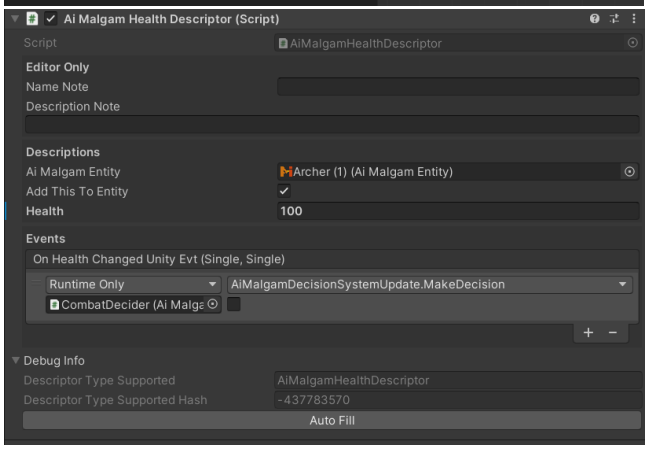


In line 12 you see a different base class that takes the required and previously selected Descriptor type as a generic argument. This is mandatory to generate the hash used in line 22 to finally retrieve the required Descriptor from the target AiMalgamEntity. Provided the target entity has the Descriptor you can check its values against the ones you may define in this special Condition node setting:

```
12  public class AiMalgamCustomConditionDescriptor : AiMalgamBaseConditionForDescriptors<AiMalgamHealthDescriptor>
13  {
14      //Add Your Check Against Values Here
15
16      Frequently called  0+33 usages
        public override AiMalgamConditionResult IsConditionMetFor(AiMalgamEntity targetAiMalgamEntity
17          , AiMalgamEntity conditionsRequesterEntity)
18      {
19          AiMalgamConditionResult conditionResult = base.IsConditionMetFor(targetAiMalgamEntity, conditionsRequesterEntity);
20
21          AiMalgamHealthDescriptor customDescriptor = targetAiMalgamEntity// AiMalgamEntity
22              .GetDescriptorsByHash<AiMalgamHealthDescriptor>(this.DescriptorTypeSupportedHash).FirstOrDefault();
23
24          //dont waste time with other checks
25          if (!conditionResult.success || customDescriptor == null)
26          {
27              //Mark failure in graph
28              this.MarkInGraph(conditionsRequesterEntity, acceptedOrFailed: false);
29              return AiMalgamConditionResult.CreateFailed(this);
30          }
31
32          //success = customDescriptor.YOURCHECK(this.CHECKAGAINSTVALUES);
33
34          //Mark success or failure in graph
35          this.MarkInGraph(conditionsRequesterEntity, conditionResult.success);
36          return conditionResult;
37      }
38  }
```

You can usually find the hashes in the respective “Debug Info” foldouts (Health-Condition & Descriptor example):

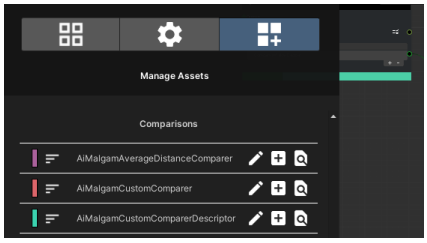


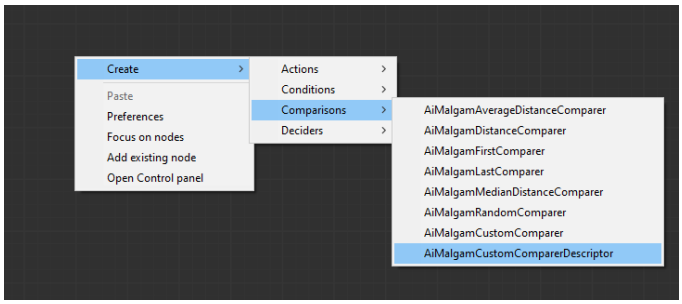


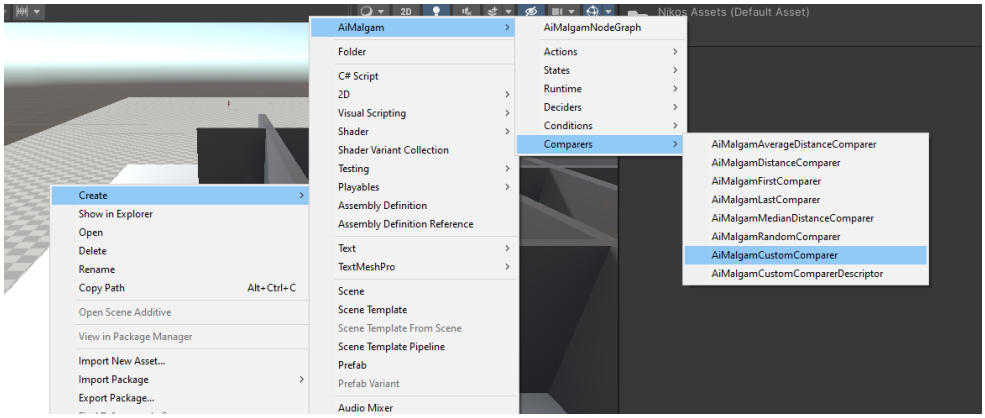
Creating new independent Comparer Node Settings

Comparer node settings are used to provide custom compare and sorting algorithms for a list of AiMalgamEntities or their required Descriptors, depending on if you check the box below. The setup is the same as in the Condition creation section.

The newly generated Comparer node settings can also be accessed via the context menus and the manage asset side panel:



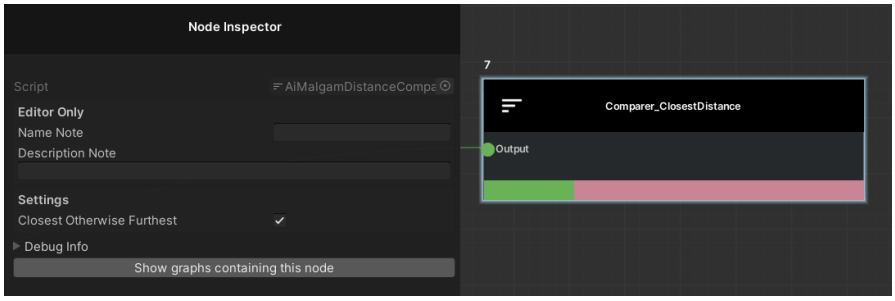




Looking into the code reveals that it contains 2 simple methods, one for the actual comparison between 2 list elements and a “global” entity (line 25) and the other one to retrieve a single entity result (line 31). You can by default remove the “GetResult()” method starting from line 31, since the “Compare()” method from line 25 is called in its base method. It is there to present you an option to temper with it:

```
12      public class AiMalgamCustomComparer : AiMalgamBaseComparison
13      {
14          /// <summary>
15          /// Compares 2 agents against each other
16          /// </summary>
17          /// <param name="entityA"></param>
18          /// <param name="entityB"></param>
19          /// <param name="checkAgainst"></param>
20          /// <returns>
21          /// -1 = B won
22          /// 0 = equal
23          /// 1 = a won
24          /// </returns>
            Frequently called  0+2 usages
25          public override int Compare(AiMalgamEntity entityA, AiMalgamEntity entityB, AiMalgamEntity checkAgainst)
26          {
27              //Compare anything here
28              return String.Compare(entityA.name, entityB.name, StringComparison.Ordinal);
29          }
30
            Frequently called  0+4 usages
31          public override AiMalgamEntity GetResult(List<AiMalgamEntity> targets, AiMalgamEntity checkAgainst, bool sortOriginalList)
32          {
33              //Add custom logic here - Otherwise you can delete this method
34              return base.GetResult(targets, checkAgainst, sortOriginalList);
35          }
36      }
```

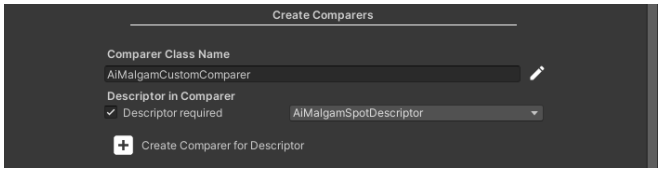
Here you can see an example Comparer node setting that gets the closest entity from the given list by comparing each entity’s distance to the 3rd parameter from line 25.



Here you can see a context where a Comparer node setting might be used. In this case to follow the closest target:

Creating new Descriptor dependent Comparer Node Settings

Similar to the Descriptor dependent Conditions you can generate Comparer settings to be also dependent of Descriptor MonoBehaviours. Just make sure to reference the additional assembly definition files if you use demo Descriptors and maintain an assembly definition file on your own.

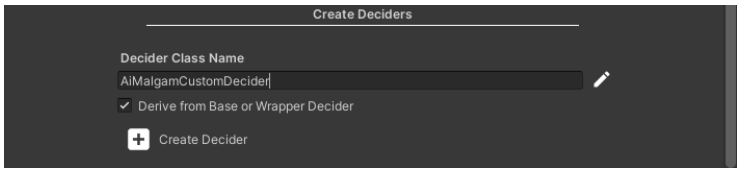


Looking into the code reveals that similar to the Condition settings a special base class is used in line 14 which takes the required Descriptor type as the generic argument. This type is then used to create the hash and to retrieve the required Descriptor from the target AiMalgamEntity as shown in line 29 and 30:

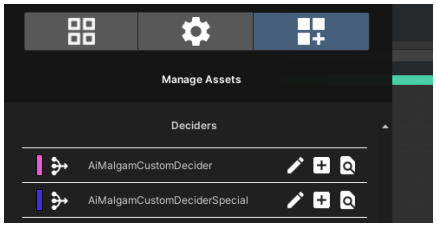
```
11      [CreateAssetMenu(fileName = nameof(AiMalgamCustomComparerDescriptor), menuName = nameof(AiMalgam) + "/"
12          + nameof(Comparers) + "/"
13          + nameof(AiMalgamCustomComparerDescriptor))]
            No asset usages  2 usages
14      public class AiMalgamCustomComparerDescriptor : AiMalgamBaseComparisonForDescriptors<AiMalgamSpotDescriptor>
15      {
16          /// <summary>
17          /// Compares 2 agents against each other
18          /// </summary>
19          /// <param name="entityA"></param>
20          /// <param name="entityB"></param>
21          /// <param name="checkAgainst"></param>
22          /// <returns>
23          /// -1 = B won
24          /// 0 = equal
25          /// 1 = a won
26          /// </returns>
                Frequently called  0+2 usages
27          public override int Compare(AiMalgamEntity entityA, AiMalgamEntity entityB, AiMalgamEntity checkAgainst)
28          {
29              AiMalgamSpotDescriptor customDescriptorA = entityA.GetDescriptorsByHash<AiMalgamSpotDescriptor>(this.DescriptorTypeSupportedHash).FirstOrDefault();
30              AiMalgamSpotDescriptor customDescriptorB = entityB.GetDescriptorsByHash<AiMalgamSpotDescriptor>(this.DescriptorTypeSupportedHash).FirstOrDefault();
31
32              //Compare anything here
33              return String.Compare(customDescriptorA.name, customDescriptorB.name, StringComparison.Ordinal);
34          }
35
                Frequently called  0+4 usages
36          public override AiMalgamEntity GetResult(List<AiMalgamEntity> targets, AiMalgamEntity checkAgainst, bool sortOriginalList)
37          {
38              //Add custom logic here - Otherwise you can delete this method
39              return base.GetResult(targets, checkAgainst, sortOriginalList);
40          }
41      }
```

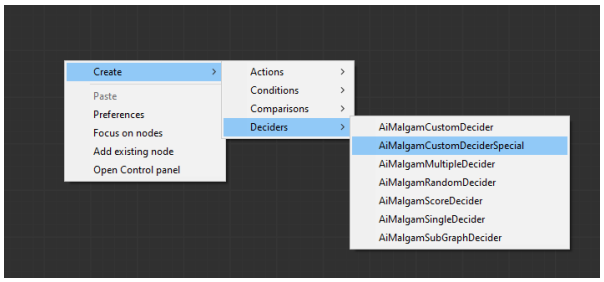
Creating new Decider Node Settings

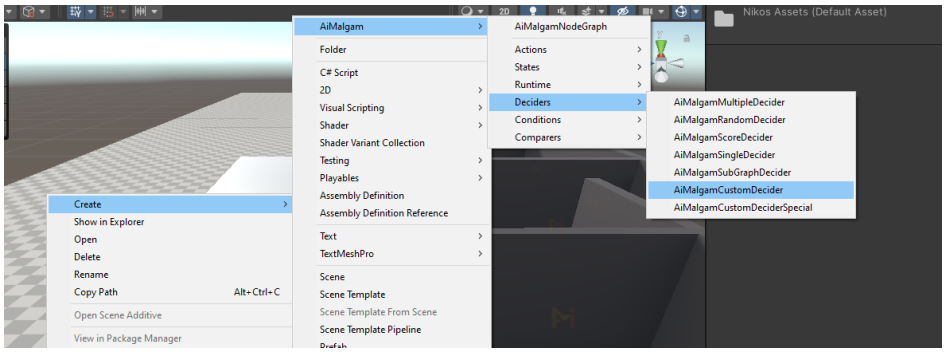
You can either create a "normal" Decider script (checkbox = true) or a "special" one that does not separate its children into leaf or branch settings, similar to the Subgraph-Decider node:



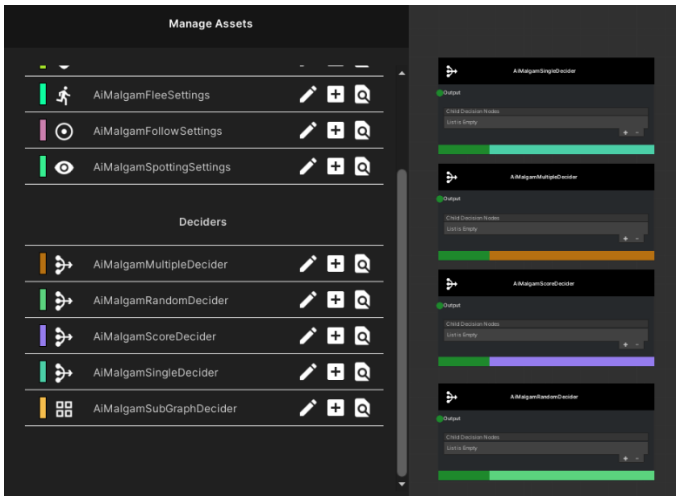
Newly generated Decider node settings can be accessed via the context menus or the manage assets side panel (as usual):







Here is a list of those “normal” Decider nodes that inherit from the AiMalgamBaseDecider class:



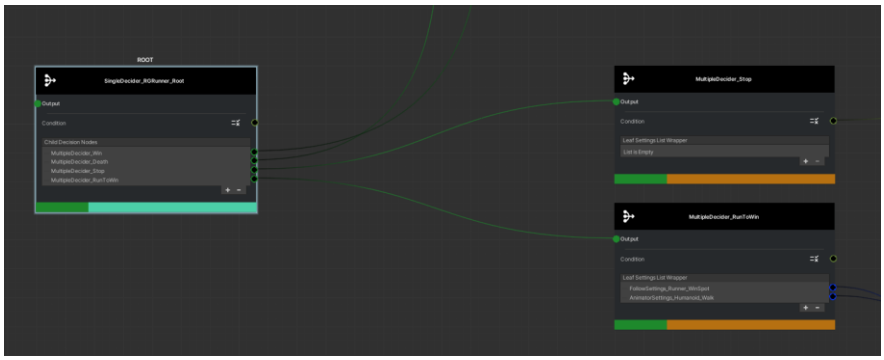
As you can see in the following illustration, a “normal” Decider setting offers 2 overridden methods to either traverse Action settings (line 48) or Decider settings (line 13). Those 2 methods are called in the **“RootTraverse()” method** found in the base class:

```
11  public class AiMalgamCustomDecider : AiMalgamBaseDecider
12  {
        Frequently called  0+1 usages
13      protected override AiMalgamDecision BranchTraverse(AiMalgamEntity aiMalgamEntity, AiMalgamDecision decision){...}
47
        Frequently called  0+1 usages
48      protected override AiMalgamDecision LeafTraverse(AiMalgamEntity aiMalgamEntity, AiMalgamDecision decision){...}
70  }
71  }
```

The provided method implements the Single-Decider traversal logic. When working with Decider children, make sure to call their **“RootTraverse()” method** as shown in line 30. Don’t forget to pass the decision parameter into it and finally return it to build the final AI decision:

```
13      protected override AiMalgamDecision BranchTraverse(AiMalgamEntity aiMalgamEntity, AiMalgamDecision decision)
14      {
15          //implement your custom branch (deciders) traversal here
16          //---example from the single decider (replace if wanted)---
17          //traverse through each child decider
18          for (var index = 0; index < this.childDecisionNodes.list.Count; index++)
19          {
20              IAiMalgamDecider foundNode = this.childDecisionNodes.list[index];
21
22              if (foundNode == null)
23              {
24                  Debug.LogWarning( message: this.name + " null child decider node at index: " + index);
25                  continue;
26              }
27
28              //determine if a new decider has been added
29              int oldCount = decision.deciderResults.Count;
30              decision = foundNode.RootTraverse(aiMalgamEntity, decision);
31              //if valid decider found, last one is the one
32              bool foundValid = decision.deciderResults.Count > oldCount
33                                && decision.deciderResults[decision.deciderResults.Count-1].success;
34
35              //we found a valid decider in the current node and dont want the other nodes and their actions to be included
36              if (foundValid)
37              {
38                  break;
39              }
40          }
41          return decision;
42      }
```

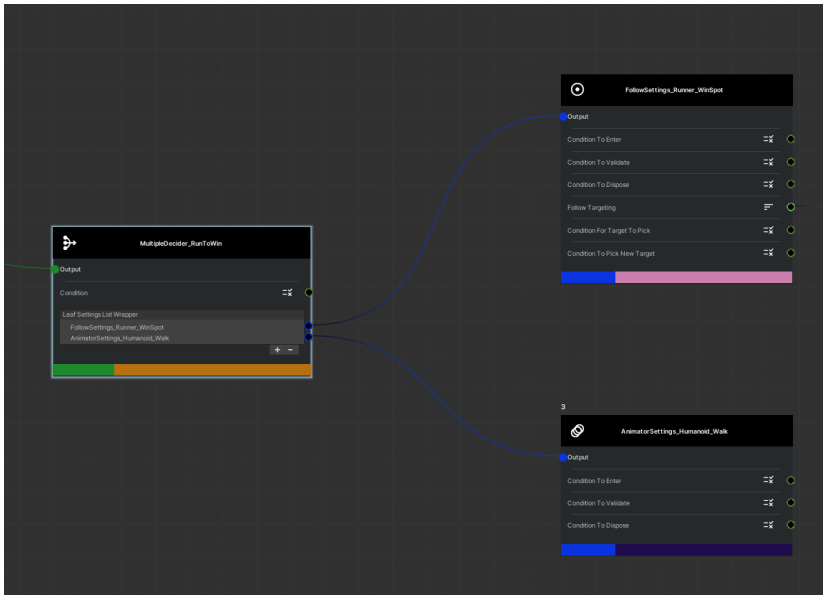
Here is an example of a “normal” branch Decider setup:



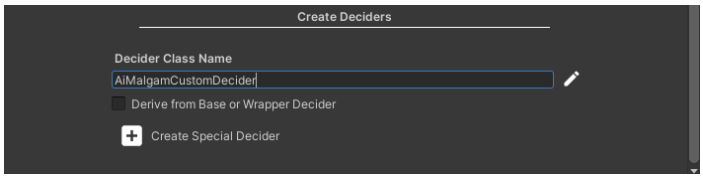
Looking into the **“LeafTraverse()” method** reveals an additional step that must be made. In line 51 you see a result object being created and assigned again at line 55, calling the base method to apply the Action setting to its respective Engine (if found) and building the result object again. Afterwards you must update the result list item (line 62) found in the decision object parameter (from line 44) to update the AI decision generally.

```
44      protected override AiMalgamDecision LeafTraverse(AiMalgamEntity aiMalgamEntity, AiMalgamDecision decision)
45      {
46          //implement your custom leaf (settings) traversal here
47
48          //---example from the single decider (replace if wanted)---
49          //define a decider result:
50          //in case there are no children, we succeed
51          AiMalgamApplyDeciderResult applyDeciderResult = new AiMalgamApplyDeciderResult(this, success:true);
52
53          foreach (AiMalgamBaseSettings setting in this.leafSettingsListWrapper.list)
54          {
55              applyDeciderResult = this.ApplyBranchLeafToEntity(aiMalgamEntity, decision, CollectionHelper.ListOfOne(setting));
56
57              //we only want to apply the first valid setting (selector tree behaviour)
58              if (applyDeciderResult.success)
59                  break;
60          }
61
62          decision.deciderResults[decision.deciderResults.Count-1] = applyDeciderResult;
63
64          return decision;
65      }
```

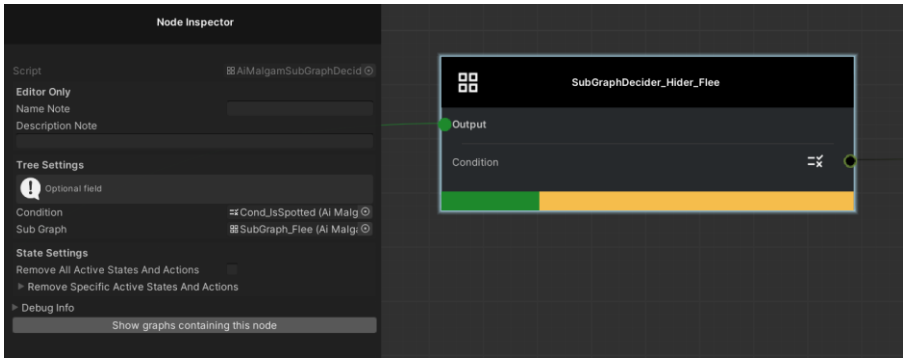
Here is an example of a “normal” leaf Decider setup:



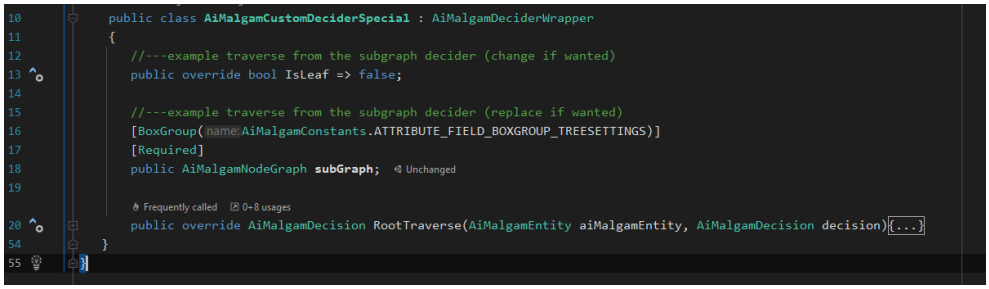
To create “special” Decider nodes that do not separate Action and Decider children, uncheck the box below it:



As mentioned, the Subgraph-Decider is such a “special” Decider:



The generated “special” Decider script reveals the Subgraph implementation and the **“RootTraverse()” method** in line 20 you can adjust to define custom traversal. Just make sure to pass and handle the AiMalgamDecision object for correct AI decision-making (tree traversal) and graph editor debugging (node color display)!



2.6.3 Custom Node Creation and understanding the Attributes

If you want to create a custom node setting that is independent of the code creator tab and other pre-defined node setting types or setup **custom input port (lists)** and make use of the other **attributes**, here is how you can do it.

To create any custom node, you can simply derive from the **AiMalgamNode** class (see API) or its sub class **AiMalgamBaseNotesScriptableObject** you can find in the API as well.

To prepare a **port list** to be **displayed in the graph** editor (for port connections), you need to create a wrapper class that derives from the ListWrapper class (line 9), specifying a type that also must inherit from the **AiMalgamNode** class in order to work. Notice, that this class must be marked **Serializable** (line 8) to be displayed in the Unity Inspector.

```
8   [Serializable]
    1 usage   1 exposing API
9   public class MyCustomNodesPortList : ListWrapper<MyCustomNodeSettings>
10  {
11
12  }
```

When working with a new custom blank node (or specializing a node setting), you must define the **output port** by offering **a field** that returns the **type of** the **hosting class**. Additionally, you must add the **[Output] attribute** above it (line 21 to 24).
Generated node settings will already contain an output port, so you don't need to do that unless you want to specify it.

Usually, you don't want to temper with that field and can hide or collapse it via the **[Foldout]** attribute in line 22 that is provided by the **NaughtyAttributes** dependency (see **Section 4**).

To display a single **input port**, you must define a field of base/ancestor type **AiMalgamNode** and put the **[Input]** attribute above it. Notice, that you can restrict the connections by specifying the type constraint (line 26 to 28).

Grouping data together can ease the maintenance and readability of settings. You can do that by using yet another NaughtyAttribute: **[BoxGroup]** (line 27 & 31).

To show the port list, as prepared above you need a field of the created wrapping type and add the [ReordableNodeList] attribute as shown in line 30 to 32. It is **very important** that the **given type matches** the **ListWrapper's type** argument you set up!

Starting from line 34 to 47 you find some interesting methods you might want to override. The **"OnOpen()" method** is called once the hosting graph editor is opened. The same goes for the **"OnClose()" method** when the currently opened hosting graph editor is closed.
When marking this node setting outside the decision's life cycle (see **Section 2.5.2**), it might be reset to white without you wanting it. You must keep that in mind and react to it by either subscribing to the selected Decision-System's events (see API) or by prohibiting the reset when overriding the **"ResetMarkingInGraph()" method** (line 44).

Another interesting method you might want to call **"OnValidate()"** is the **"NotifyOnNodeChanged()"** method. It notifies any listener of this node setting (like Actions) to changes made.

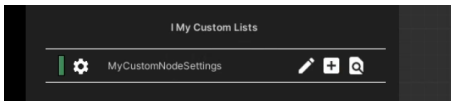
```
16  [CreateAssetMenu(fileName = nameof(MyCustomNodeSettings), menuName = "AiMalgam/MyCustomNodes/" + nameof(MyCustomNodeSettings))]
17  [NodeIcon( resourcePath: "settings")]
18  [CreateNodeMenu("AiMalgam/MyCustomNodes/")]
        5 asset usages   6 usages   2 exposing APIs
19  public class MyCustomNodeSettings : AiMalgamBaseNotesScriptableObject, IMyCustomList
20  {
21      [Output]
22      [Foldout( name: AiMalgamConstants.ATTRIBUTE_FIELD_FOLDOUT_DEBUGINFO)]
23      [ReadOnly]
24      public MyCustomNodeSettings output;   Unchanged
25
26      [Input(typeConstraint: TypeConstraint.InheritedAny)]
27      [BoxGroup( name: "MyBoxGroup")]
28      public MyCustomNodeSettings input;   MyCustomNodeSettings_2.asset
29
30      [ReordableNodeList(typeof(MyCustomNodeSettings))]
31      [BoxGroup( name: "MyBoxGroup")]
32      public MyCustomNodesPortList inputPortList;   Serializable
33
            Frequently called   0+2 usages
34      public override void OnClose()
35      {
36          base.OnClose();
37      }
38
            Frequently called   0+3 usages
39      public override void OnOpen()
40      {
41          base.OnOpen();
42      }
43
            Frequently called   0+4 usages
44      public override void ResetMarkingInGraph(AiMalgamEntity requester, bool forceRepaint = false)
45      {
46          base.ResetMarkingInGraph(requester, forceRepaint);
47      }
```

The port type constraints explained:

```
/// <summary> Tells which types of input to allow </summary>
55 usages   niggo1243 *   9 exposing APIs
public enum TypeConstraint
{
    /// <summary> Allow all types of input</summary>
    None,

    /// <summary> Allow connections where input value type is assignable from output value type (eg. ScriptableObject --> Object)</summary>
    Inherited,

    /// <summary> Allow only similar types </summary>
    Strict,

    /// <summary> Allow connections where output value type is assignable from input value type (eg. Object --> ScriptableObject)</summary>
    InheritedInverse,

    /// <summary>
    /// Allow connections where output value type is assignable from input value
    /// or input value type is assignable from output value type
    /// </summary>
    InheritedAny
}
```

If you want to create your custom node from within the graph editors side panel, you must implement a custom interface that implements the IAiMalgam interface:

```
public interface IMyCustomList : IAiMalgam
{

}
```



To create your custom node from the project window's context menu, you must add the [CreateAssetMenu] attribute (see Unity's API) above the class header. Note, that this attribute will **NOT be inherited**!

```
[CreateAssetMenu(fileName = nameof(AiMalgamRandomDecider), menuName = nameof(AiMalgam) + "/"
    + nameof(Deciders) + "/"
    + nameof(AiMalgamRandomDecider))]
```

To create your custom node from the graph editor's grid area, you need to add the [CreateNodeMenu] attribute. This attribute **will be inherited** and you just need to define the context **menu path** and **not** the **file name**!

```
[CreateNodeMenu("Create/Deciders/", order: 100)]
```

To define a special width in pixels within the grid area, use this attribute:

```
[NodeWidth(500)]
```

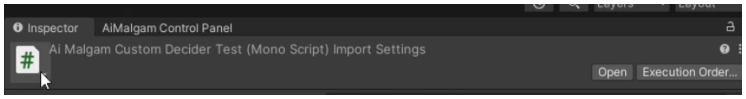
If you want to always show or always hide your custom nodes in the grid area, add this attribute above your class header:

```
[NodeAppearance(nodeAppearance: NodeAppearance.Minimum)]
```

```
public enum NodeAppearance
{
    //display only core/ relevant nodes
    Minimum = 0,

    //display some more or less relevant nodes
    Average = 1,

    //display all nodes
    Redundant = 2
}
```

To add an icon in the asset viewer, side panel and the grid area, add the following attribute above your class header. You can either set a **resource path** or a **Texture2D directly**. Note, that this is **not** the **same icon displayed in the project's window**. For that you need to go to the script file and change the icon in the top left corner.

```
[NodeIcon( resourcePath: "decision")]
```



2.7 Example AI Behavior Implementations

Note, that the **AiMalgam demo implementations** are **custom expansions** from the core system and **not directly part of it**, meaning that **you can build** and implement a (**better**) **version** yourself 😉!

The current example implementations consist of:

- Spotting behavior
  - Contacts (Trigger & Collisions)
  - Raycasts
  - View Cones (One, Hals and 4 Angle setups)
  - Distance
  - React to spotted entities differently
  - Filter spotted entities into separate containers
- Follow behavior (Rigidbody, transform and Navmesh)
  - Tailing
  - Just looking
  - Flanking
  - Randomly roaming around
- Flee behavior (Rigidbody, transform and Navmesh)
  - Flee at random point (away) from chase direction
- Animation behavior
  - Set animation parameters of any time
  - Dynamic (external) or fixed values
  - Blend and damp float values over time
- Combat behavior
  - Define a custom combat procedure (prepare, attack, cooldown, etc.)
  - Work with combat tools
  - Compatible with animation events and animation or physics driven combat
- Any behavior
  - You can quickly create simple AI Actions by inheriting from this Action implementation
- Special Conditions and Descriptors used in the Demo environment

Since the custom demo AI behavior implementations would blow up this document and can be quite complex to understand at first glance, I advise you to look into this video.

3. Planned Features & Fixes

- A decision statistic showing how often and where the traversal is done
- Showing connections on both sides to improve the visibility in the graph editor
- Toggling certain connection types in the graph editor
- As far as possible, improve the graph editor performance
- Improve and fix small issues in the demo setups by exchanging animations and working on the existing AI setups
- Add search fields for nodes in the Control Panel and graph editor
- Add filter and sorting mechanics for the graph editor nodes list section
- Don't clear undo/redo when deleting AiMalgam assets
- Implement an FPS limiter/ stabilizer option to load balance and reduce AI calls, rather than slowing down the machine
- Add fuzzy logic examples
- Add goal-oriented behavior (GOB) examples
- Add sound AI behavior examples
- Add networking AI behavior
- Add Boids AI examples
- Add "daily life" AI examples (like in a SIMS or farmer game)
- Add 2D examples
- [Add **YOUR** ideas and feature requests (or bugs) by contacting me 😉]

4. Dependencies

See the **"Third-Party Notice.txt"** file for more information about the third-party licenses.

Unity3DHelperTools (developed and maintained by this author):
An open-source collection of handy tools.
If you use assembly definition files, add this to make use of this framework:

- NikosHelpersAssembly.asmdef

Alternatively, you can download this pack from the asset store (for free).

NaughtyAttributes forked from dbrizov:
An open-source editor extension framework to organize the Inspector. This version has improved performance for large data files.
If you use assembly definition files, add this to make use of this framework:

- NaughtyAttributesPerfFork.Core.asmdef

xNode originally developed by Thor Brigsted:
An open-source node graph editing framework **highly modified** and **improved** to fit this asset pack. Since this modified version is built mainly to work with AiMalgam, it might not work out of the box for custom node-based implementations that derive from it directly. Feel free to try It out by adding this assembly definition file to yours:

- XNodeCustomN.asmdef

The 1st and 2nd tool you can also import via the **manifest.json** file, found in **(Package/manifest.json)**. With that procedure, no direct import of the dependency packages is required and can be made possible with the following example configurations:

```
{
    "scopedRegistries":
    [
        {
            "name": "NaughtyAttributesPerfFork",
            "url": "https://upm-packages.dev",
            "scopes": [
            "com.nikosassets.naughtyattributes"
            ]
        }
    ],
    "dependencies"
    {
        "com.nikosassets.u3dhelpertools":
"https://github.com/niggo1243/Unity3DHelperTools.git#upm"
    }
}
```

Or:

```
{
    "scopedRegistries":
    [
        {
            "name": "NaughtyAttributesPerfFork",
            "url": "https://upm-packages.dev",
            "scopes": [
            "com.nikosassets.naughtyattributes"
            ]
        },
        {
            "name": "Unity3DHelperTools",
            "url": "https://upm-packages.dev",
            "scopes": [
            "com.nikosassets.u3dhelpertools"
            ]
        }
    ],
    "dependencies"
    {
        "com.nikosassets.u3dhelpertools": "1.1.1"
    }
}
```

5. FAQ & Troubleshooting

Q:
Why can't I select or modify the input field of a (newly created) node setting in the graph's node Inspector side panel!?
A:
This is a yet unknown Unity issue (still investigating) that pops up at random. You can just reopen the graph editor and it should work again!

Q:
The Animation-Actions and Action settings are not working. What's the problem?
A:
Make sure to add the Animator-Engine to your animated entity and configure the animator to use the same animator controller the setting refers to. Using "Apply Physics" as the "Update Mode" might cause problems in the animator controller as well. Also apply the animator to the animator list in the Animator-Engine. Otherwise make sure that the Action was applied successfully in the first place and check its life cycle settings!
In some other cases the animator controller might get stuck in a state, or you tried to change the animation during an animation transition. Also observe if the desired animator parameters were unset/ reset, before the animation (transitions) had a chance executing in the controller.

Q:
Why can't I add a node setting to the graph's grid area?
A:
Check if the node already exists in the graph and look at the warnings in the console. Also make sure that the graph editor's visibility is set to display your placed node (might be added but hidden).

Q:
I tried to change the AI decision tree during play mode, but it didn't change the behavior. What's the issue?
A:
Changing connections during play mode is not recommended and can cause issues. You can change the settings at any time but not the connections at runtime.

Q:
Why didn't the node colors update after I changed them in the preferences?
A:
Reopen the graph 😉

Q:
Why are all materials pink?
A:
You probably use another render pipeline. See **Section 2.1** to fix that issue.

Q:
Why can't my entity spot anything?
A:
Make sure to add Colliders to your entities and check the layer mask, as well as the allowed layer mask in the Spotting settings. Also check what spotting type you are using. If it is contact, make sure to add the Contact Collector MonoBehaviours and assign them in the Spotting-Engine!
Also in some cases when using raycasts or view cone spotting, the spotting might be blocked by an unwanted or misplaced Collider (make sure to assign the correct caster transform in the Spotting-Engine to not hit the floor for example). This Collider can either be located on your entity or the target, or somewhere in the scene (some invisible triggers?)!

Q:
Why aren't the fancy Inspector settings rendered correctly?
A:
This can be caused by other Inspector extensions, like Odin or custom ones inside another asset pack you are using. There is no general solution for that. You need to find the other Inspector extensions to proceed with a solution.

Q:
Why are no combat hits applied to the target?
A:
Make sure the target has the Combat Descriptor attached to it and the AiMalgamEntity references it (checkout the "Debug Info" foldout). Also, when using collision-based hit appliers, make sure to add Rigidbodies to the target and the hit appliers as well!
Furthermore, check if the "can hit" checkbox of a hit applier or if you are not using that, the CombatTool is checked during the hit incident.

Q:
Why is my entity's combat stuck sometimes?
A:
When using animation events to proceed with certain CombatStateProperty setups, you might want to check if your animation got interrupted at some point, not emitting the desired event that continues your combat. You need to keep multiple things in mind:
Animation controller and its transitions, the Animation-Actions being called at some time, the Combat-Actions being called at some time, your CombatTools and if they are allowed to apply the next CombatStateProperty. The animation parameters might also be reset before they could be executed in the animator controller.

Q:
I can't find the AiMalgam classes and namespaces in my code. Where are they?
A:
If you are using assembly definition files, make sure to reference the required ones (see **Section 2.1**).

Q:
Why are the newly added input ports not selectable and are not displaying correctly?
A:
Should theoretically not happen, but in case it does: Reopening the graph should fix this.
In case this happens, please inform me with your reproduction steps!!

Q:
My Deciders did succeed or fail even though I expected the other result. What's going on?
A:
Check the Decider settings in the Inspector and investigate its fields. Also check if your Action settings validate or invalidate at certain moments, like if the stack limit was reached or your entity was in a certain State.

Q:
Why is my entity's movement pattern so weird?
A:
If you are using look at in one Action and actual movement in another that uses the "Translate" movement method, switch to "Transform Apply" instead and setup the blending settings to smooth the apply transform movement.

Q:
Why does my entity jitter while moving?
A:
If you set up an animator with root animation movement checked and also provide a NavMeshAgent and a Rigidbody, make sure that you choose the right "Interpolate" setting (e.g. "None") in the Rigidbody and the right "Update Mode" (e.g. "Normal") in the animator.

Q:
Why do the "mapped" float values not return expected results?
A:
It is important that the raw "input" values you want to convert/ map to are within the pre-defined min-max input bounds!
Otherwise the math won't work out.

Q:
Why is my entity moving so slow?
A:
The movement speed of a transform setup with a NavMeshAgent and an animator is different in some Unity versions, especially starting from 2021.0. You can adjust the speeds on the "MalgyNavMeshSpeedAdjustment" MonoBehaviour if you work with the existing prefabs and they happen to include this component. It is also recommended to not set "Animate Physics" as the "Update Mode" of the animator component!

Q:
Why are some of my connections lost?
A:
First, did you save your progress last time?
Did Unity crash at some point?
Did you temper with connections during play mode?
Did you change any port field or its attribute (contents) in the script files?
Although this should not happen, a possible solution is to reset your commit (when using git) and deleting the library folder (You can leave the shader cache, or anything related to visuals or audio).
If this happens, please contact me with your reproduction steps so this issue can be terminated!!

Q:
Why is Unity importing assets forever?
A:
It seems that there is a corrupt ScriptableObject in your project. You can back up your project and slowly delete the ScriptableObjects for each new start to locate the corrupt one. Also delete the library every time you want to test the project!
Also keep the **asset file names** and (global) **folder depths as short as possible**, since Unity might have a problem reading **too long strings/ filepaths**.

If you can't find your problem here, feel free to contact me and we can extend this section for other future Developers in need!

6. Changelog

AiMalgam 1.1.0

Fixes:
- Fixed too slow and jittery movement for the humanoids (Malgy) by adjusting the NavMeshAgent speeds depending on the Unity version and disabling animated physics
- Animations sometimes got stuck or not play for archers and paladins (Animation controller fixes)

Changes:
- Now displaying the required Descriptors for certain Conditions in a graph to run correctly on a decision system
- Improved and modified the manual (FAQ, Changelogs and small adjustments)
- Updated the HelperTools dependency to version 1.1.1

AiMalgam 1.0.2

Changes:
- Removed third-party animations and replaced them with custom ones (walk and run)
- Added the "Third-Party Notice.txt" license file
- Small modifications in the manual pdf

AiMalgam 1.0.1

Fixes:
- Issues in the (multiple) decider pathing and debug display
- Debugging graph not repainting/updating when the selection changed

AiMalgam 1.0.0

- Initial release!

7. Contact & Support

If you need any help or have issues regarding this package, feel free to contact me at:
nikos.assets@gmail.com

Store page:
https://assetstore.unity.com/publishers/52812?preview=1

And please consider reviewing this package 😊!