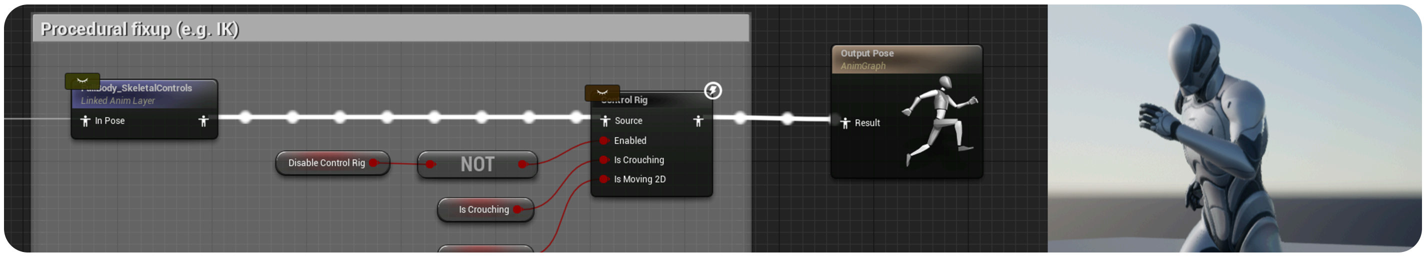
Graphing in Animation Blueprints

Edit, blend, and manipulate poses on Skeletal Meshes using various graphs in Animation Blueprints.



The primary method of using **Animation Blueprints** is by creating logic in the **Anim Graph** and **Event Graph**. This logic then defines the pose behavior, variables, and other properties of the Blueprint. These graphs work together to provide the final output pose of the character at any given frame.

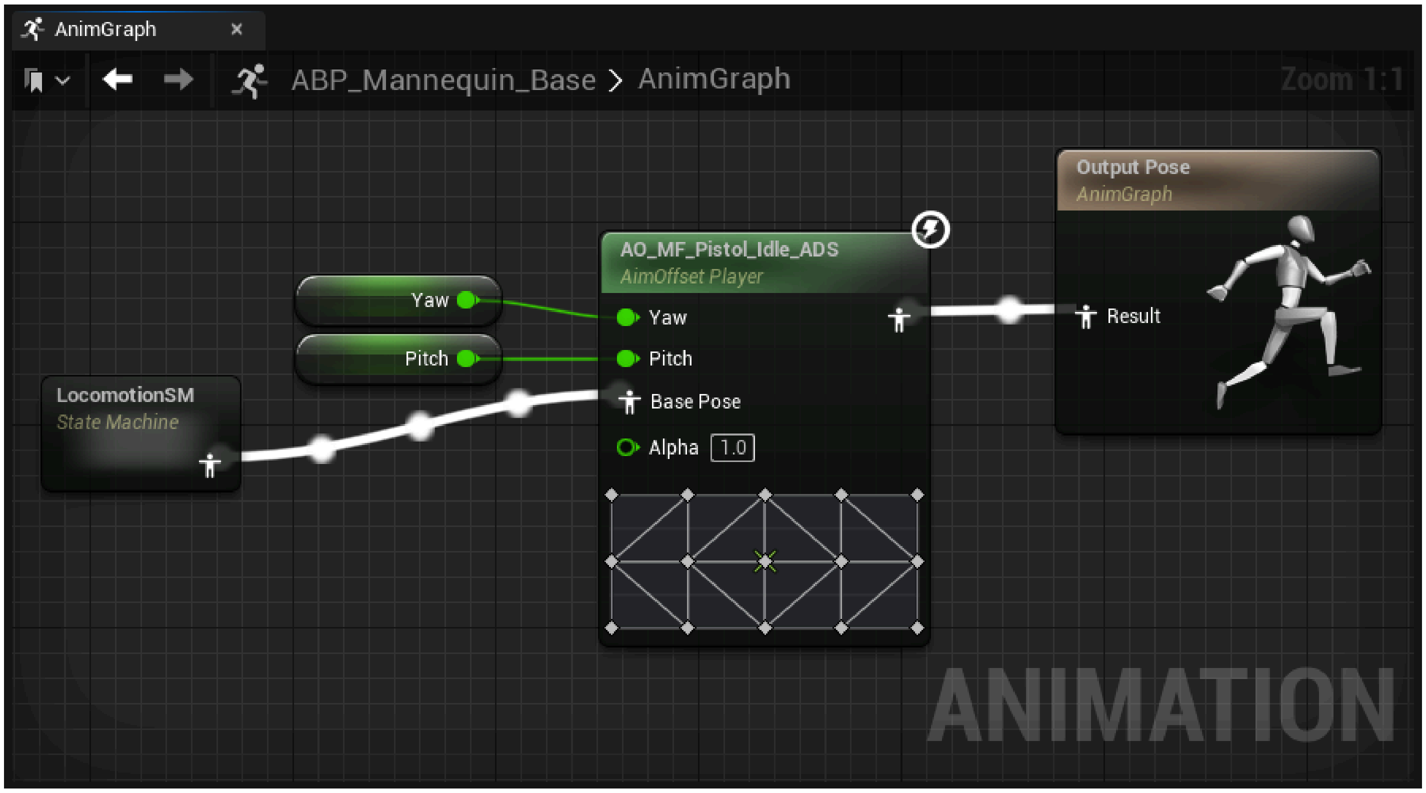
This document provides an overview of the Anim Graph, Event Graph, and the graphing experience in Animation Blueprints.

Prerequisites

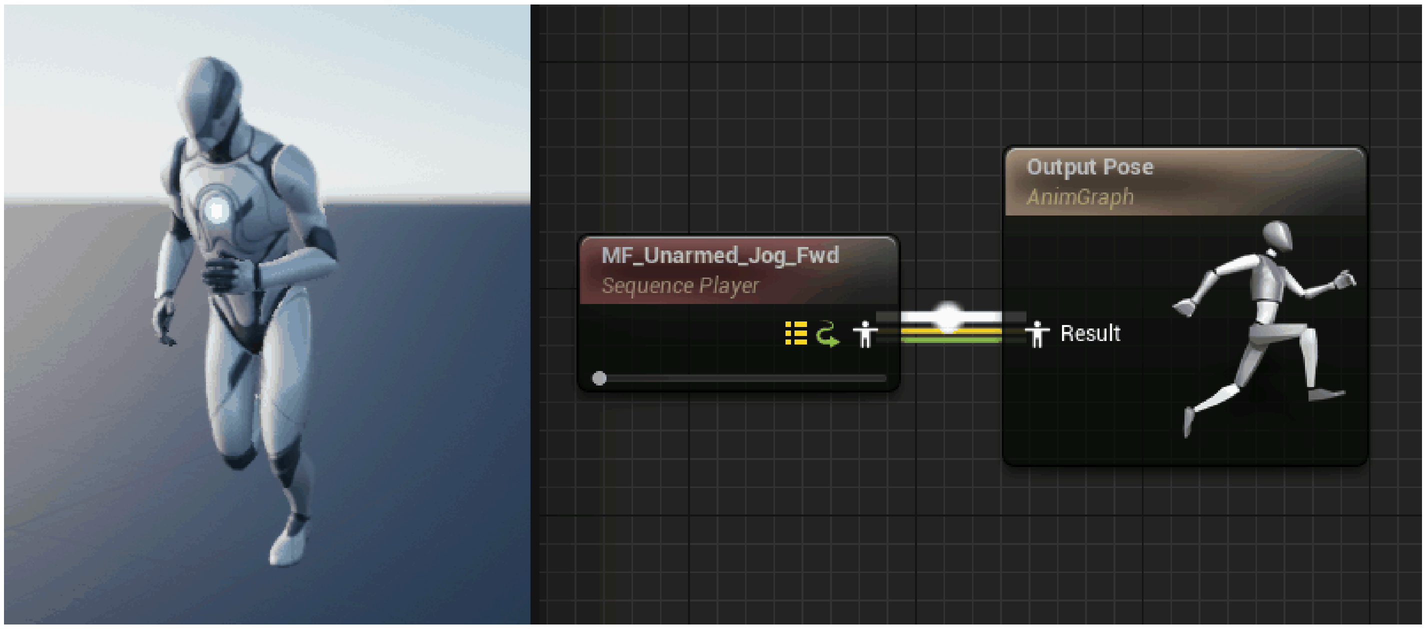
- You have created an Animation Blueprint and opened it.
- You have a basic understanding of Blueprint Visual Scripting, from which the Animation Blueprints derives its interface and behavior.

Anim Graph

The AnimGraph is where you create animation-specific logic for your character. Typically, this includes creating nodes which control blending, transforming bones, locomotion, and other similar animation effects. Inside the AnimGraph, you can use values calculated from the EventGraph, or Functions, and then connect those variables as inputs for your AnimGraph nodes, such as Blend Spaces. The combined effects of your nodes and their values are connected to the **Output Pose**, which is where the final pose of the character is defined for every frame.

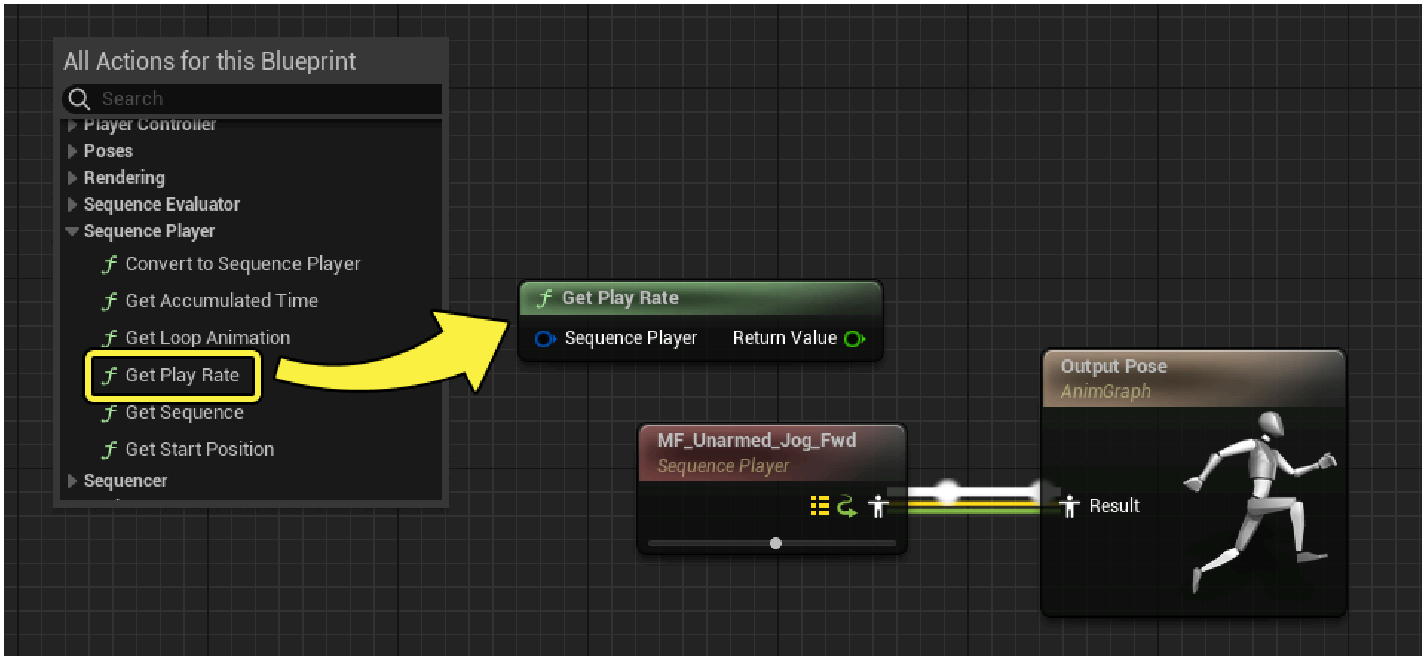


For a basic example, you can create a **Sequence Player** node in the AnimGraph, which references an Animation Sequence. Connect this node to the **Output Pose** node, then click **Compile** in the Toolbar. You should now see your animation play indefinitely, as the **Sequence Player** continuously plays the animation.

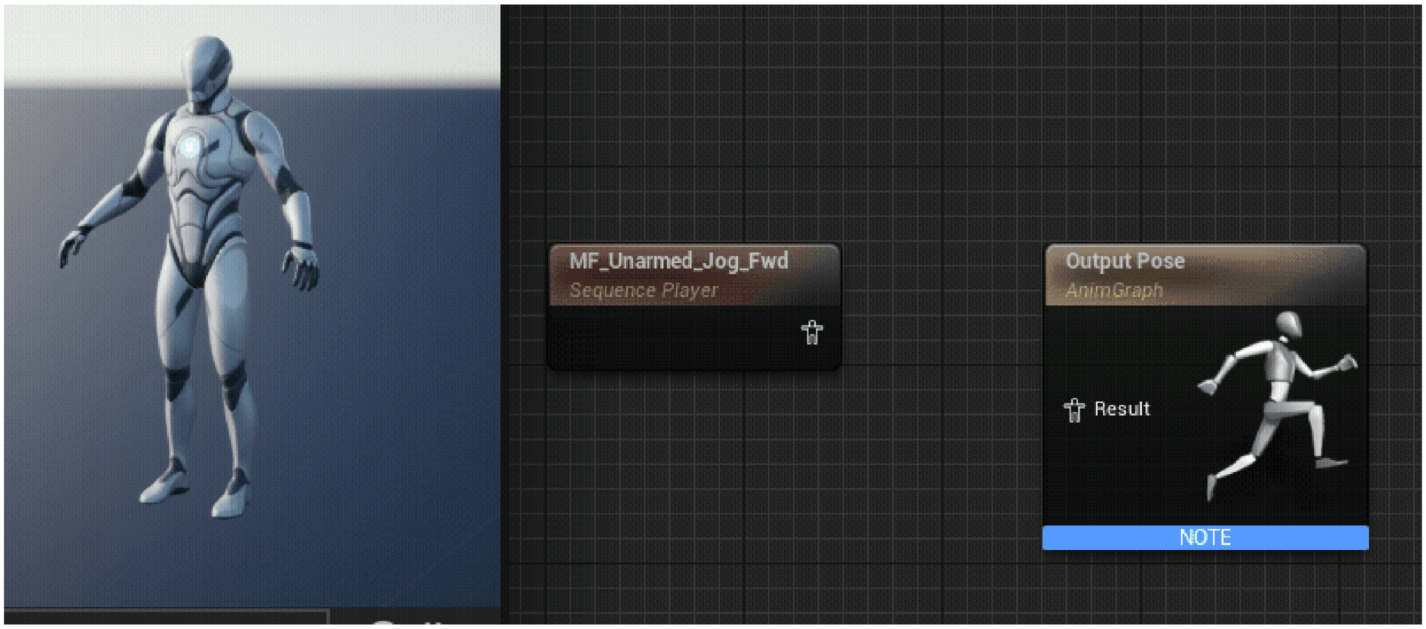


Creating and Connecting Nodes

Similar to Blueprints, nodes are created by right-clicking in the graph and selecting a node.



If you create a node that outputs **Pose** information, then this node can be connected to other pose pins, such as on the **Output Pose** node. Typically, creating and connecting nodes will require you to recompile the Blueprint.



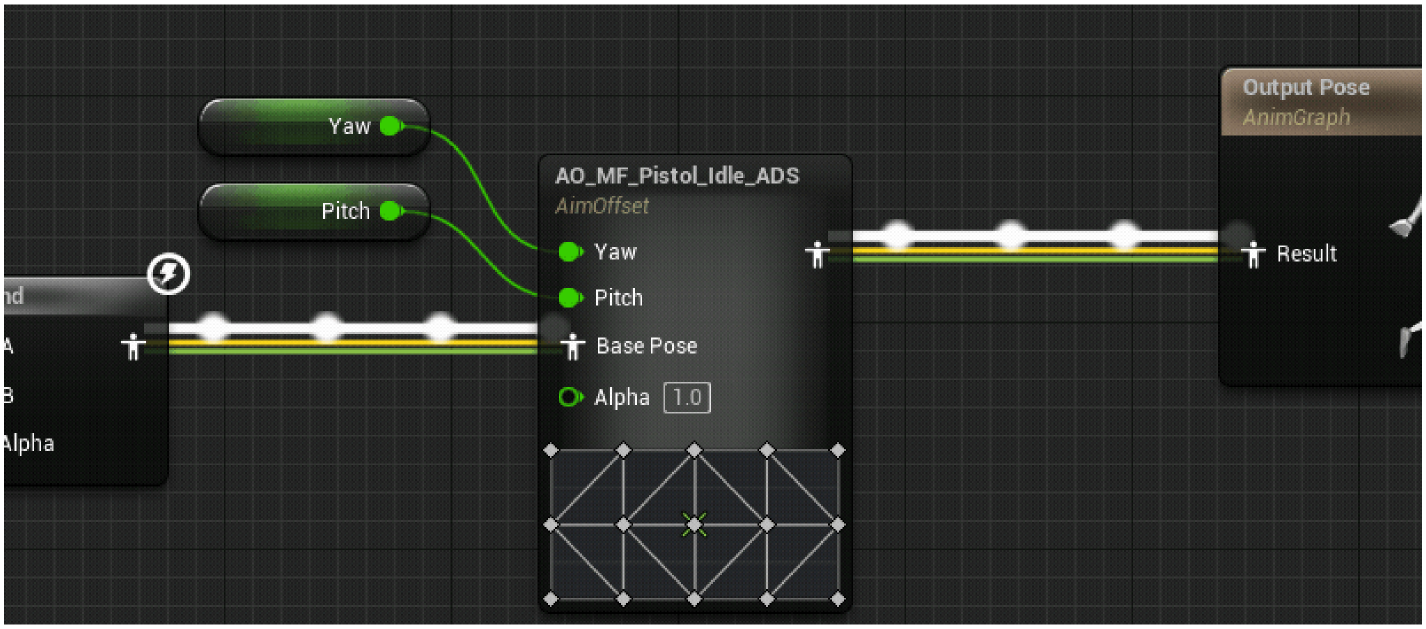
There are several node types you can create in the AnimGraph. To learn more about them, refer to the Animation Node Reference page.

Animation Node Reference

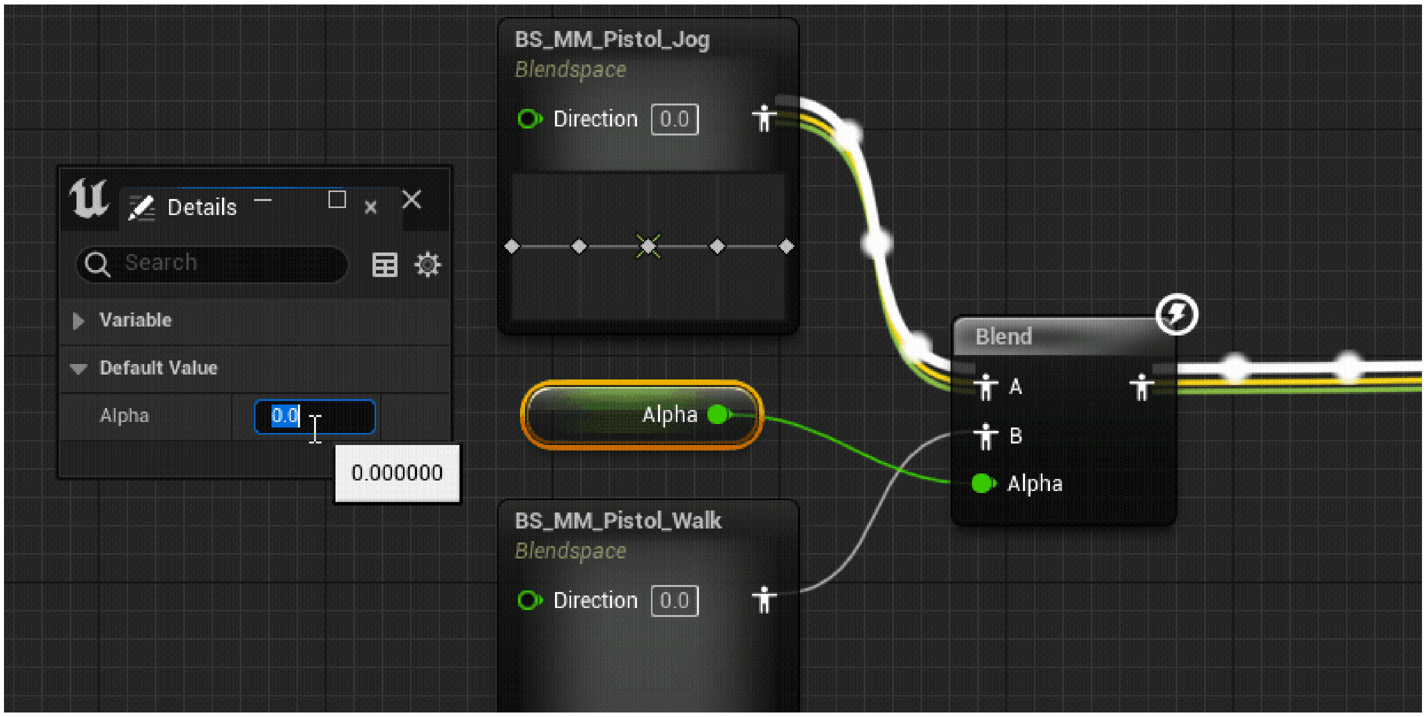
Descriptions of the various animation nodes available for use in Animation Blueprints.

Execution Flow and Information

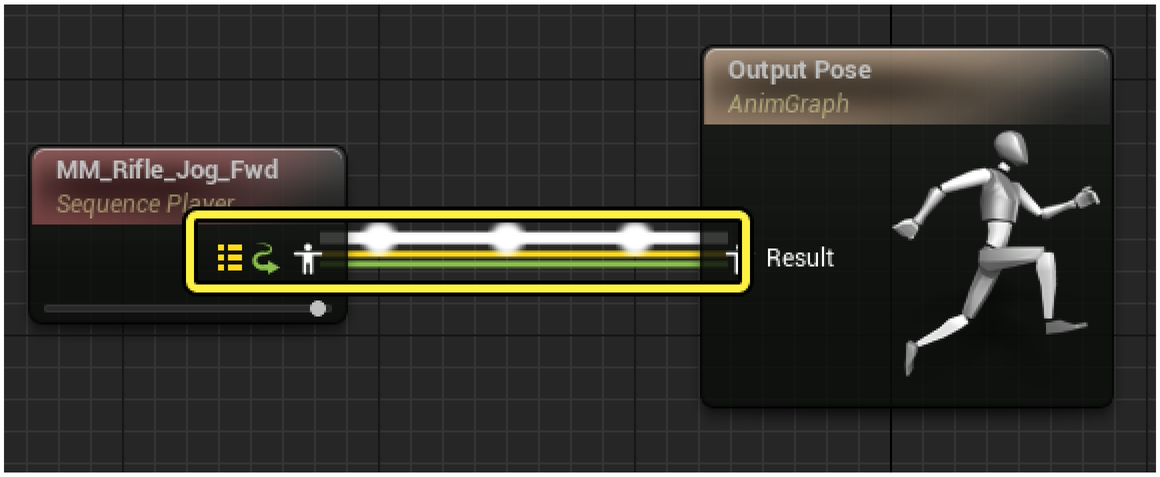
Pose connections, which are denoted with the **Pose icon**, have their execution shown as pulsing links along their connection lines. For regular graphs, such as the EventGraph of a Blueprint, this flow is visualized during play as it is dependent on the firing of events. The AnimGraph is different in that it displays the flow of execution at all times since it is not event-based and is evaluated each frame.



In the AnimGraph, the flow of execution represents poses being passed from one node to another. Some nodes, such as **Blends**, have multiple inputs and make a decision internally on which input is currently part of the flow of execution. This determination is usually dependent on some external input, like the value passed to a data pin. In this example, the **Alpha** value on a Blend node is being set to either **0** or **1**, which enables or disables the evaluation of each incoming pose.



Poses and nodes can also contain several internal attributes, which are represented by parallel execution lines between the connected pins and icons on the node. This information conveys additional attributes that are being sent along with the animation pose.



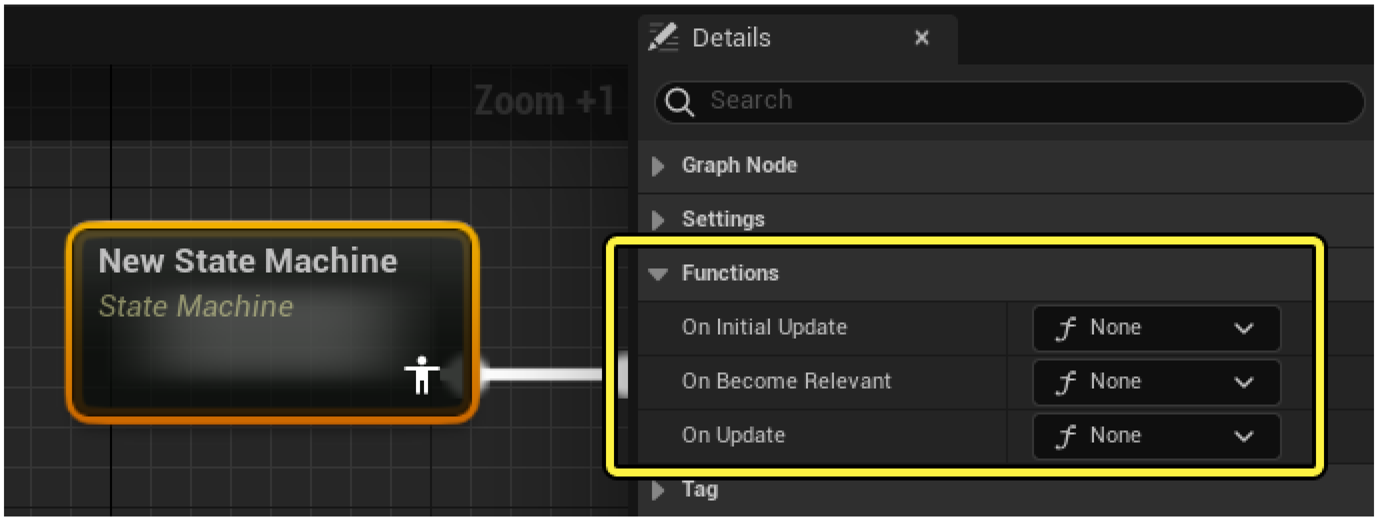
The list of attributes are as follows:

Attribute	Icon	Description
Curves		Passes Anim Curve data.
Attributes		Passes Animation Attribute data.
Sync		Passes Sync Group data.
Root Motion		Passes Root Motion data.
Inertial Blending		Passes Inertialization data. This indicator only appears when the Inertialization node is requested, typically when a blend occurs.

Node Functions

Alongside the normal node execution in the AnimGraph, you can also specify functions to call during a node's execution steps. This makes it possible to create more manageable logic that is organized within its relevant node and function. Additionally, this saves on CPU resources so that these events are only called when a node is active

These function properties can be accessed by selecting any **animation node** in the **AnimGraph** and locating the **Functions** property category in the **Details** panel.



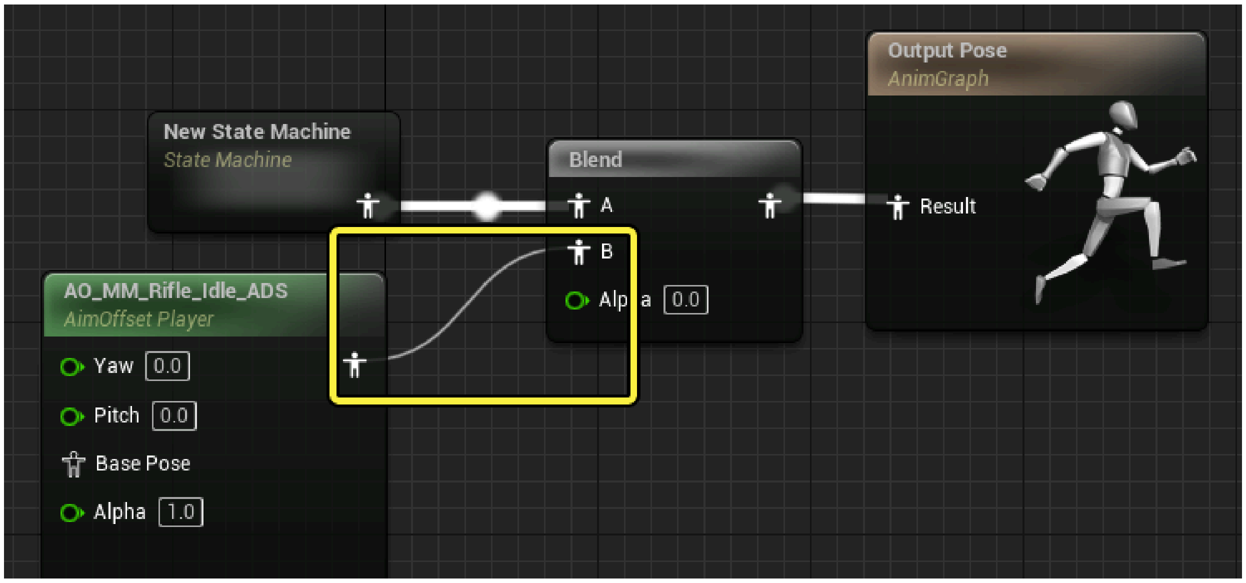
On Initial Update: None
On Become Relevant: None
On Update: None

The following Function properties are available:

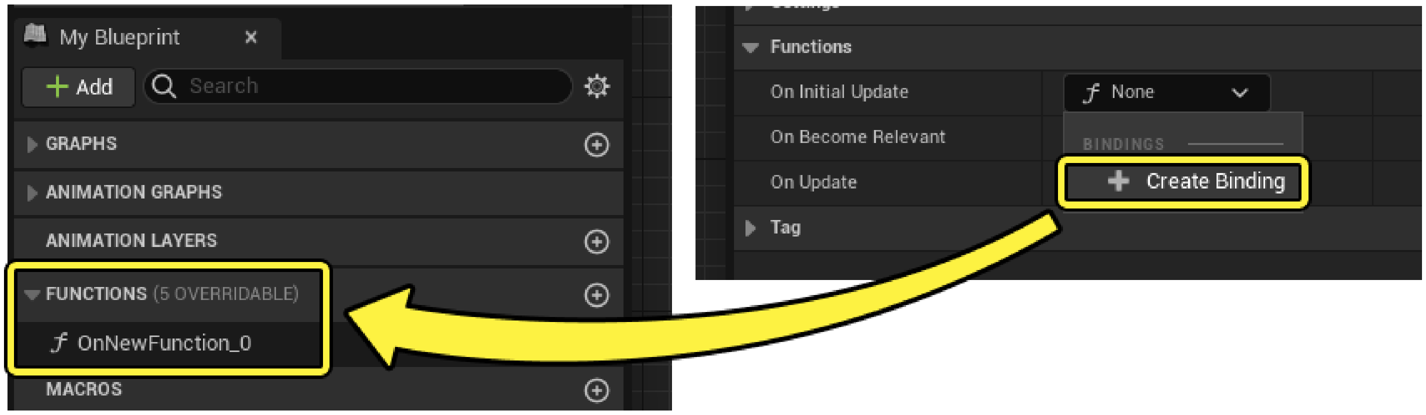
Function Type	Description
On Initial Update	This Function will execute only once during gameplay or simulation. It executes the first time the node becomes relevant, and before **On Become Relevant**. It does not re-execute if the node becomes relevant repeatedly.
On Become Relevant	This Function executes the first time the node becomes relevant, but after **On Initial Update**. It will also re-execute if the node becomes relevant repeatedly, such as if it were to be blended on, then off, then on again.
On Update	This Function executes continuously every tick as long as the node is relevant. It starts executing after **On Become Relevant** executes.

The concept of **relevance** in the AnimGraph refers to if a node is being evaluated or not. In cases when nodes are not being evaluated, such as when using blend nodes or state machines, some nodes may be completely inactive. When this occurs, the node is **not relevant**. Only nodes currently contributing to the Output Pose are considered **relevant**.

In this example, the Aim Offset node is not relevant because the Blend node is blended completely to input **A**.

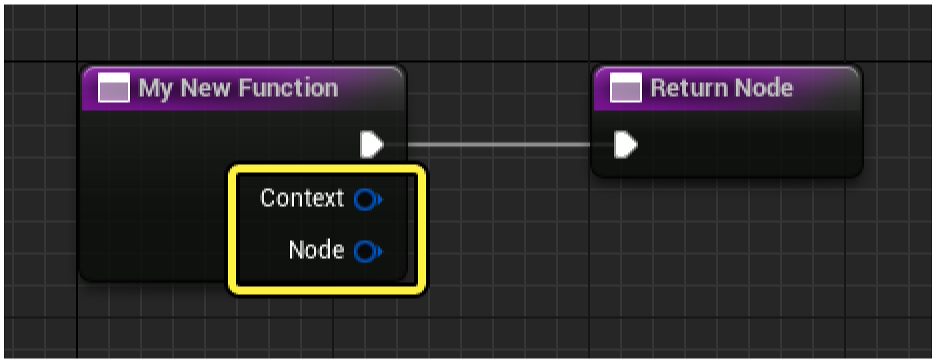


To add a new **Function**, click the **dropdown menu** for your chosen **property** and select **Create Binding**. This will create a new function for your Animation Blueprint and bind it to the Function property.

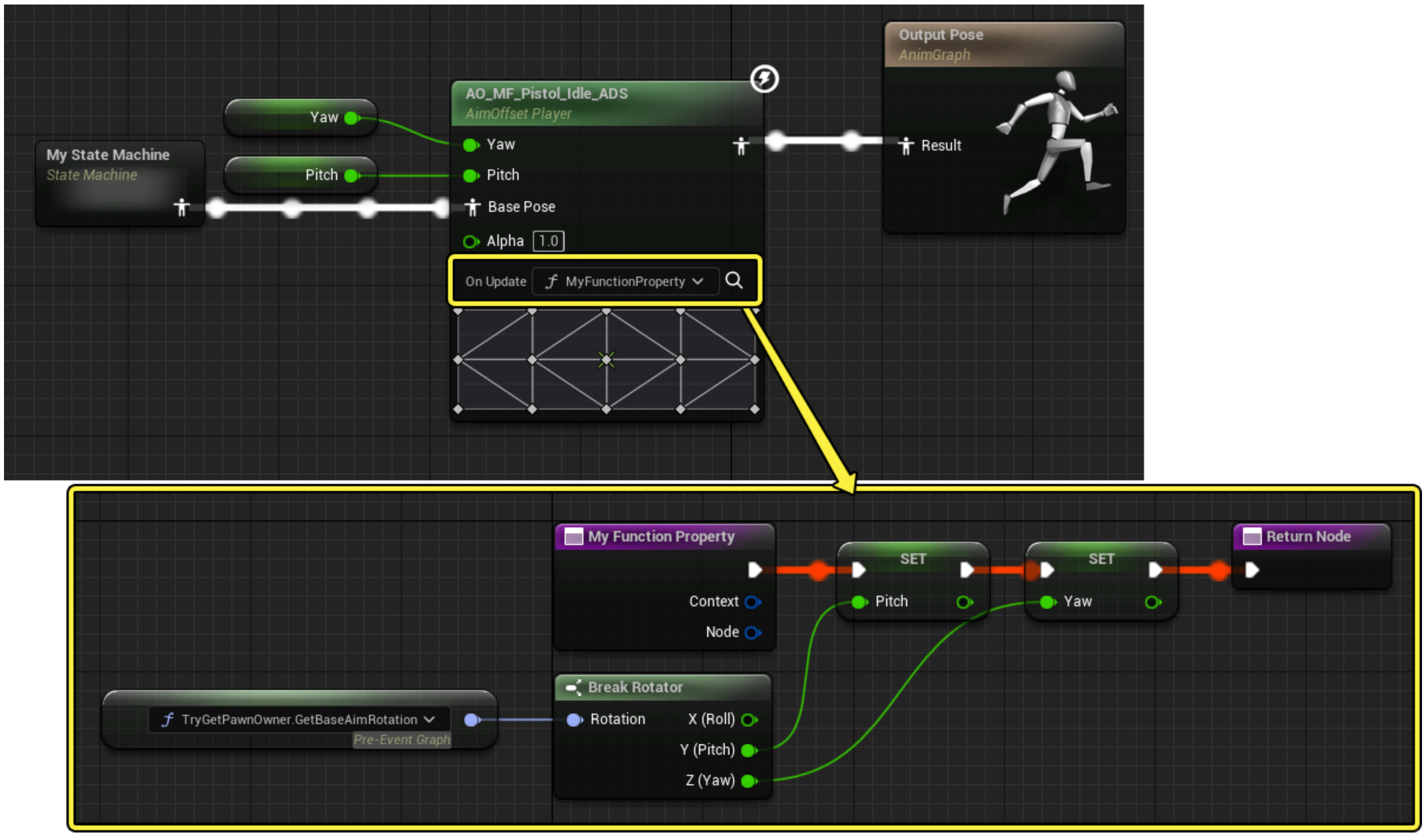


When you create a function this way, it will automatically create input pins which are used to link the function to the node it's associated with. These pins are optional in some cases, but are required if you are using the function to read the current state of the node.

- **Context**: Allows the node to pass data through that is relevant to the node, such as Delta Time, or Inertialization requests.
- **Node**: Allows the node to pass itself through this pin. Typically you will want to convert this pin to a specific node type using a *Convert* function.



When a function is assigned to a node, this is indicated by the function name being visible on the node. In this example, the Aim Offset logic, for getting a character's rotation and setting the pitch and yaw values, is all contained within the function. This logic executes only while this node is being updated, rather than permanently as it would if it was in the Event Graph, reducing performance costs.



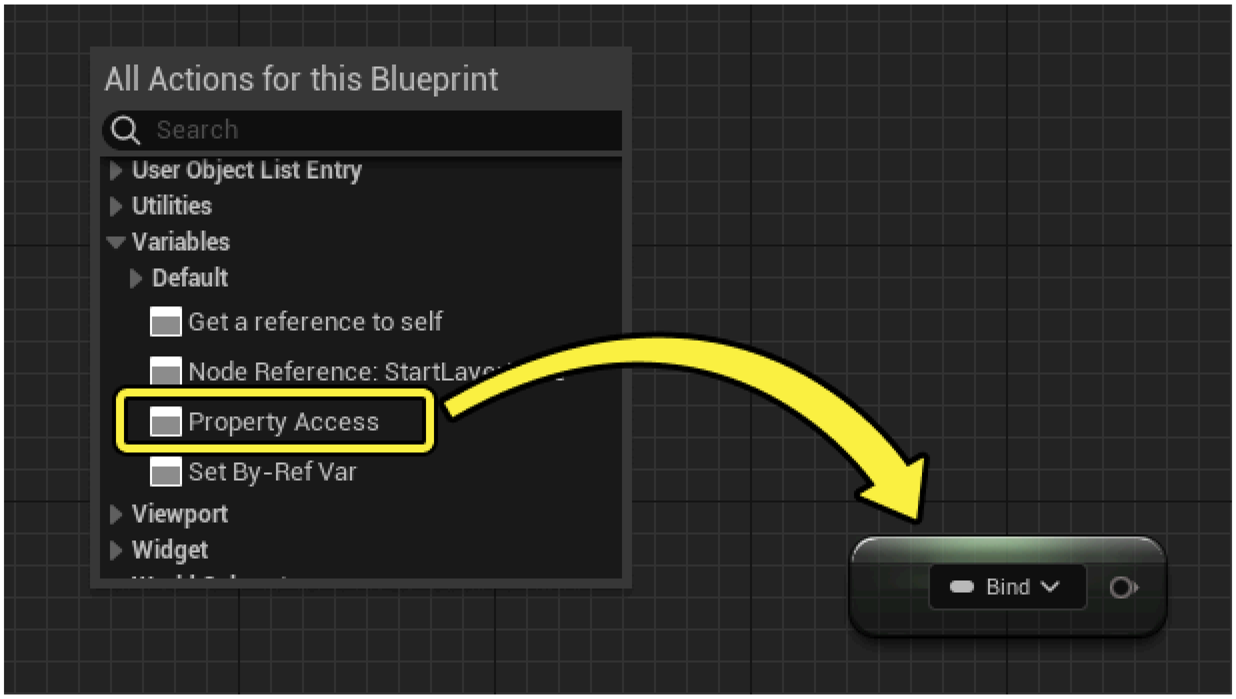
Property Access

In order to expedite getting and setting properties, you can use the **property access** feature, which contains a variety of automated functions. Property access is useful for reducing the instances of getting properties, redundant connections, and overall graph complexity. It also provides a way to automatically provide gameplay data to your animation graphs in a thread-safe way.

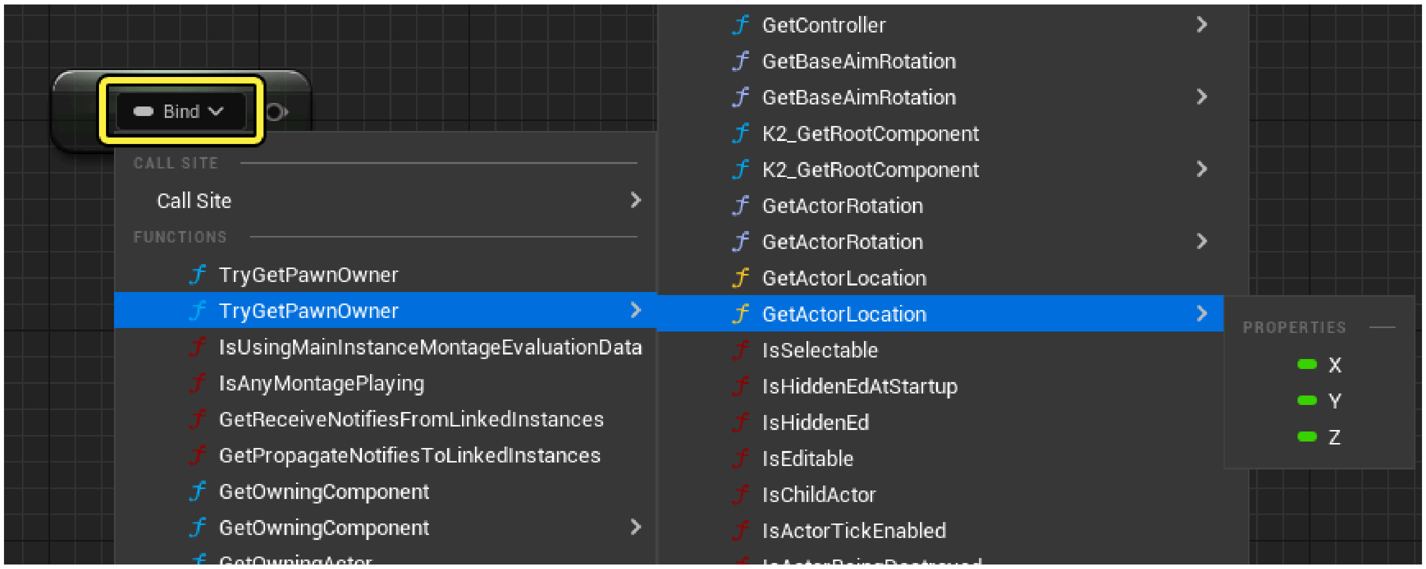
There are two main ways to use this feature:

As a Node

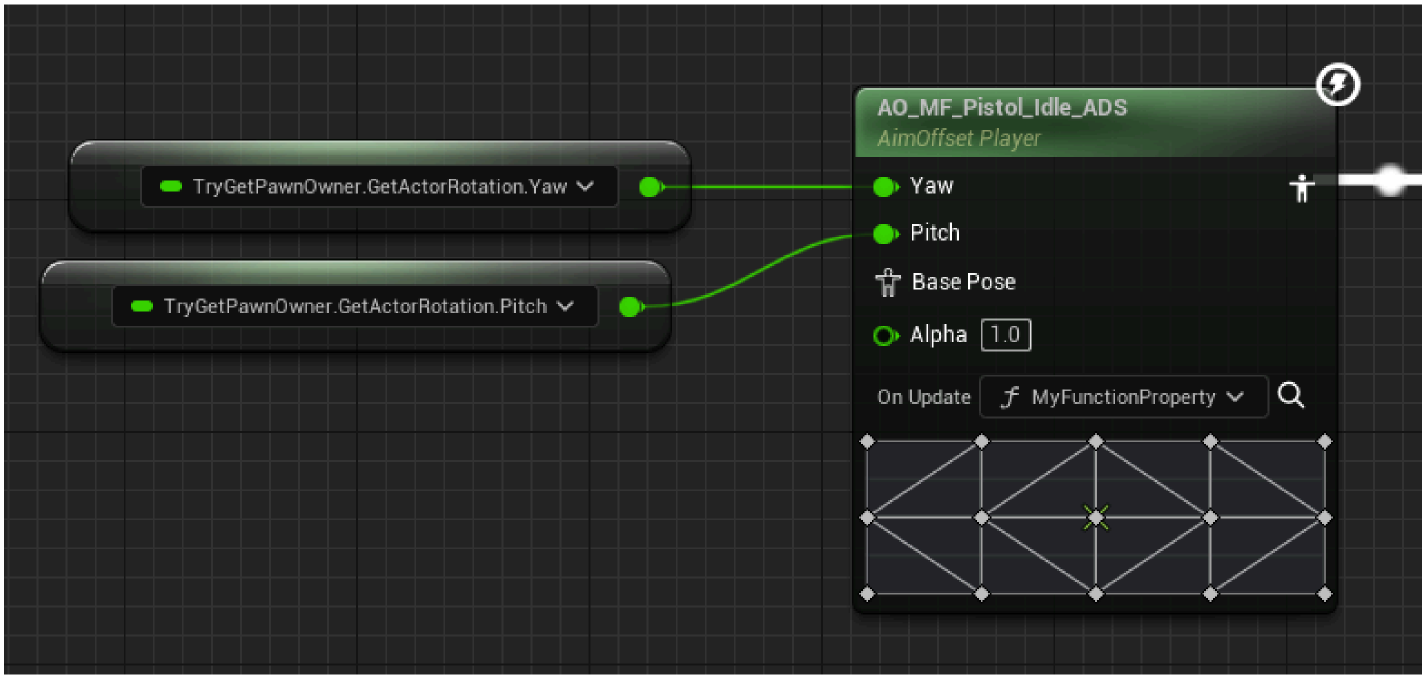
To create a **property access node**, right-click in the **AnimGraph** and select **Property Access** from the **Variables** category.



Once added, you will need to bind it to a **Get** function. Click the **drop-down menu** on the node and select the **Get function** you want. You can also navigate beyond a single Get, and locate a more specific property. In this example, a Get property path is created from **TryGetPawnOwner**, to **GetActorLocation**, to a **specific axis**.

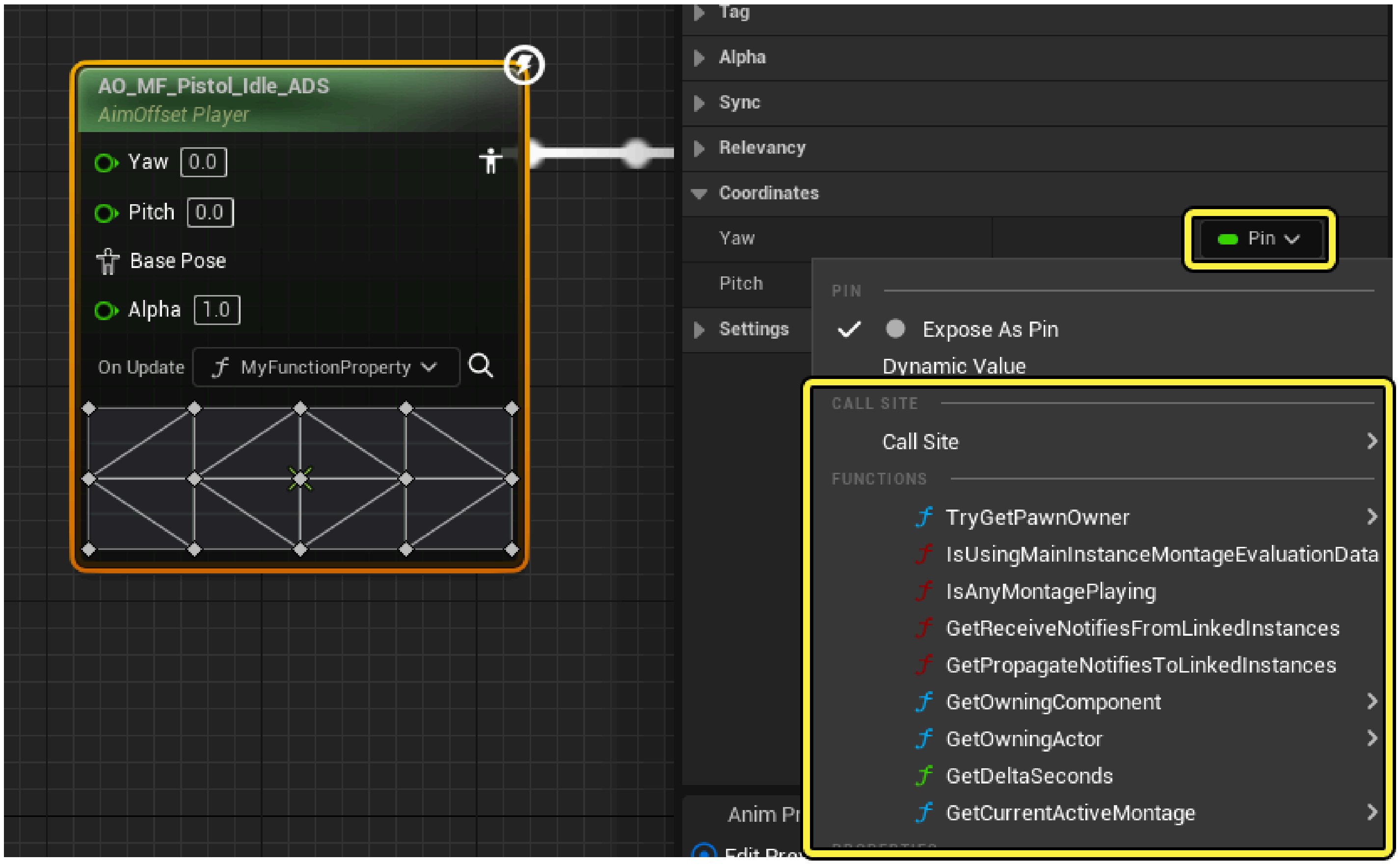


Once bound, you can use the property access node to provide property logic in your graph.



As a Pin

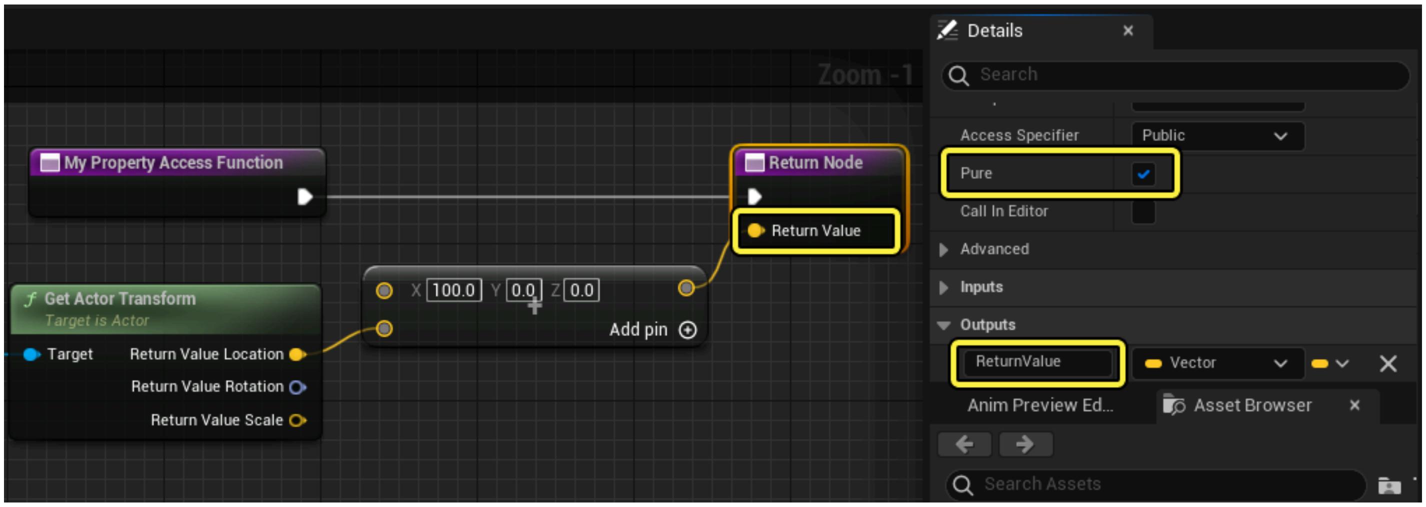
For simpler logic, you can also embed property access directly onto a property pin. Select the **node**, locate the **pin** property in the **Details** panel, and click the **dropdown menu** for that property. From here you can select a similar **Get function chain** to map to this node's property.



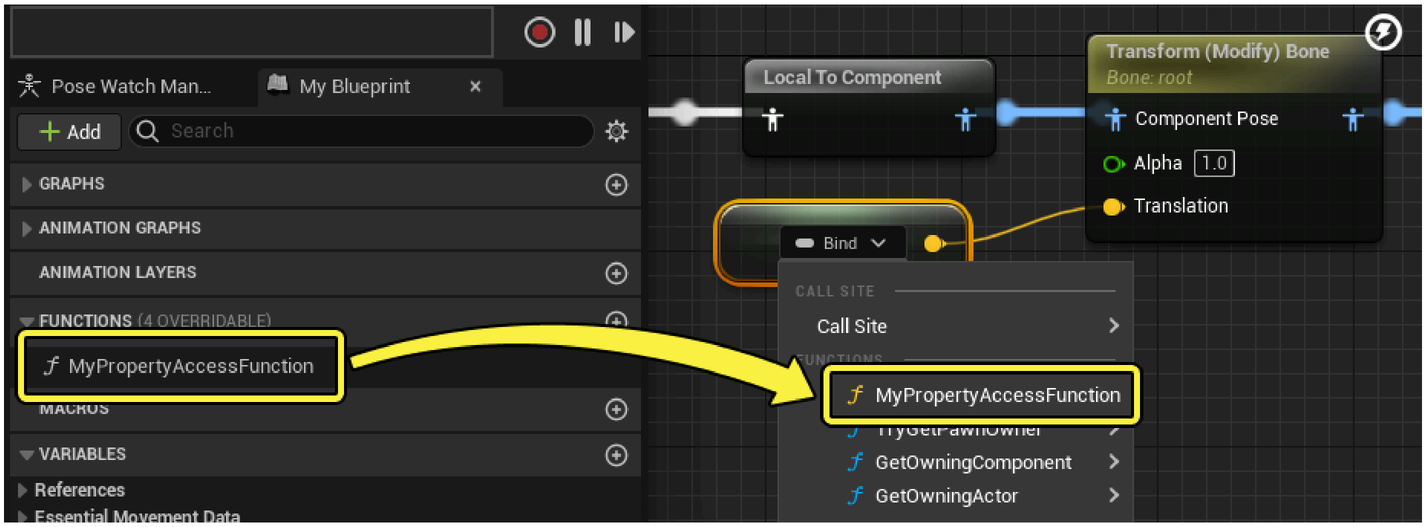
Property Access Functions

Custom functions can also be created to output values that require more complex logic. In order for the property value to output correctly, the following must be done on the function:

- The function must contain a **Return Node** with the final resulting property connected to a **ReturnValue** output pin.
- **Pure** must be enabled from the function's **Details** panel.

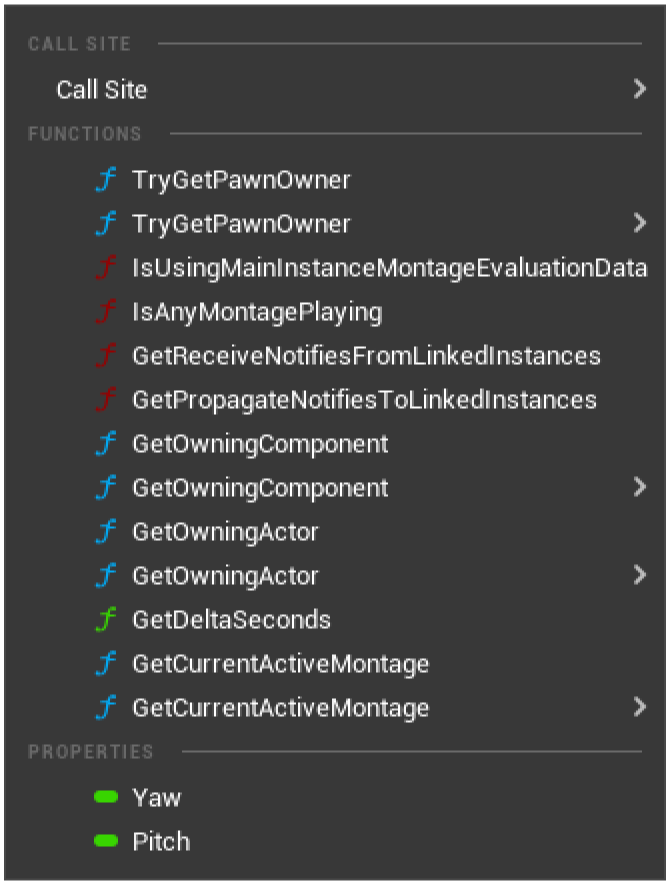


Once created and properly set up, these functions can be added from the Property Access menu.



Property Access Settings

The property access menu contains the following selections and properties:



Name	Description
Call Site	The Call Site controls which CPU thread to execute the property access on. You can select the following options: • **Automatic**, which will automatically determine the call site for this property access based on context and thread safety. You should leave this as the selection in most cases. • **Thread Safe**, which will execute this property access on the Worker Thread. • **Game Thread (Pre-Event Graph)**, which will execute this property access on the Game Thread, before the Event Graph executes. • **Game Thread (Post-Event Graph)**, which will execute this property access on the Game Thread, after the Event Graph executes. • **Worker Thread (Pre-Event Graph)**, which will execute this property access on the Worker Thread, before the Event Graph executes. • **Worker Thread (Post-Event Graph)**, which will execute this property access on the Worker Thread, after the Event Graph executes.
Functions	The list of functions you can bind to the property access.
Properties	The list of variables in your Animation Blueprint, as you can also bind property access to variables.

CPU Thread Usage and Performance

When creating complex AnimGraph logic, it may become necessary to consider performance and costs of the graph logic. By default, the AnimGraph executes on a separate CPU thread from the EventGraph, which is referred to as the "Worker Thread". The "Game Thread" is the CPU thread where the Event Graph of the Animation Blueprint and all other Blueprints execute instead.

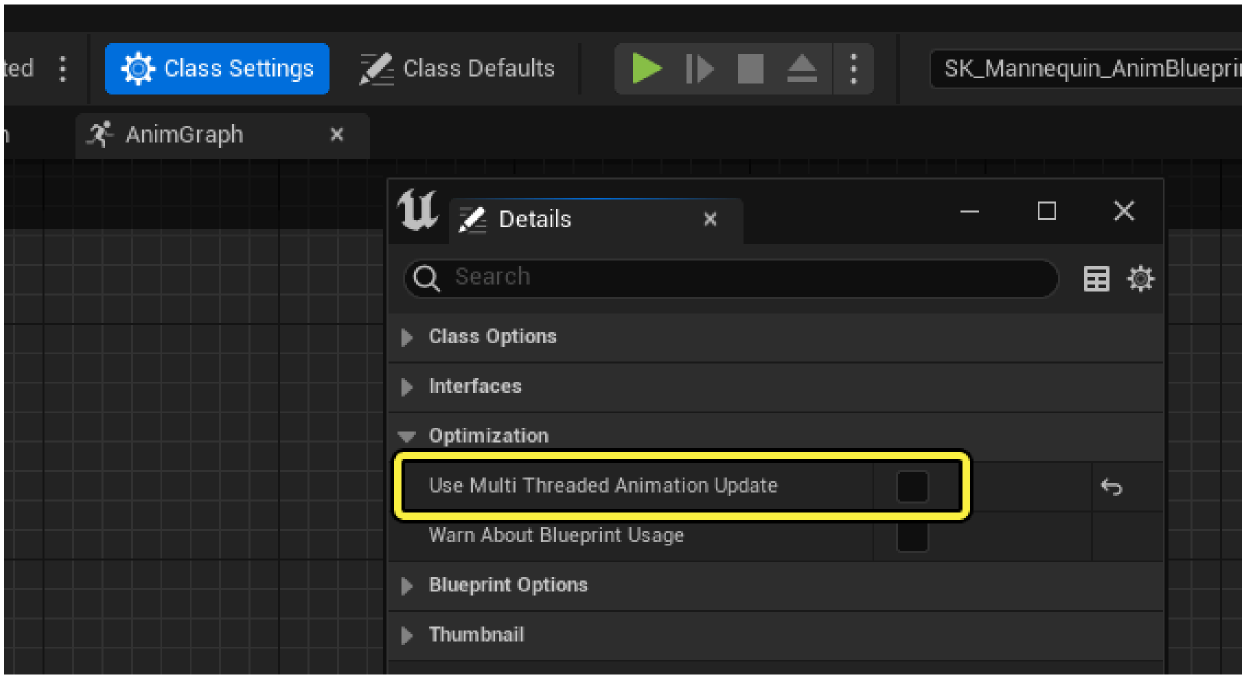
This behavior significantly improves the performance of games on multi-core machines by allowing animation work to complete in parallel with other updates.

The compiler will also warn if unsafe operations are performed in the AnimGraph. Safe operations generally are:

- Using most AnimGraph nodes.

- Directly accessing member variables of your Animation Blueprint (including members of structures).
- Calling Blueprint pure functions that don't modify state (such as most math functions).

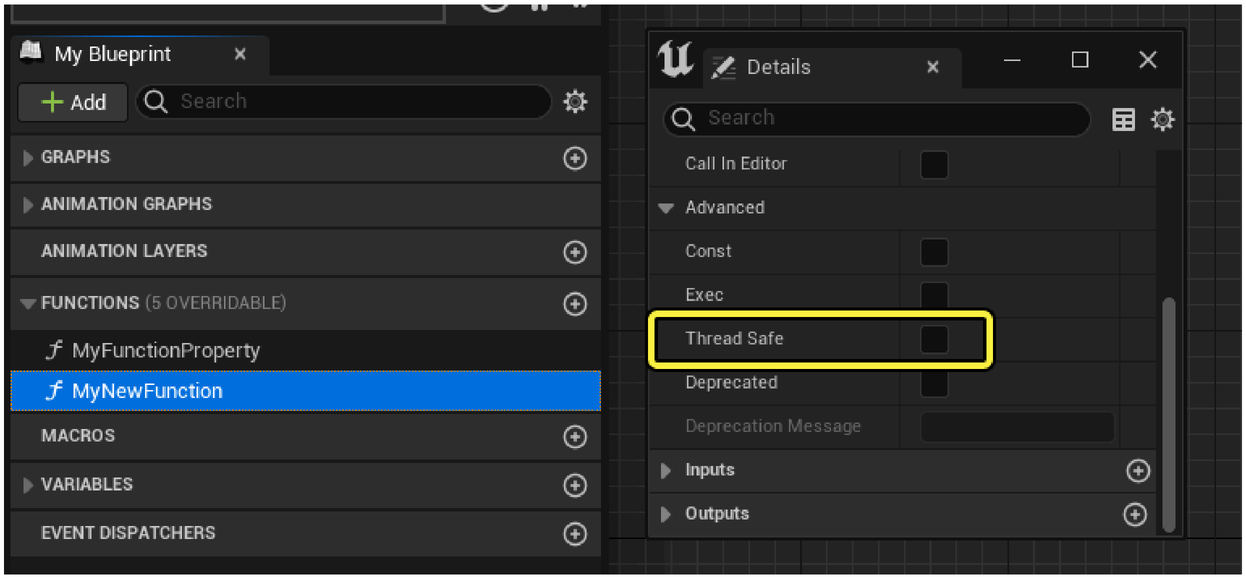
Although we do not recommend it, you can disable this behavior so that the AnimGraph executes on the game thread by disabling **Use Multi Threaded Animation Update** from the **Class Settings Details** panel.



Use Multi Threaded Animation Update
Warn About Blueprint Usage

In addition to the AnimGraph, Functions can also optionally execute on the Worker Thread. Then, when using Functions in conjunction with Node Functions, you can offload your entire Animation Blueprint graph to run exclusively on the Worker Thread, further improving performance.

Although we do not recommend it, you can disable this behavior so that Functions execute on the game thread by disabling **Thread Safe** from your selected Function's **Details** panel.



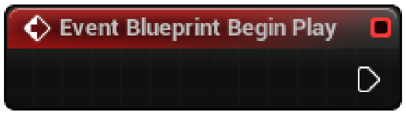
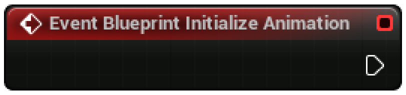
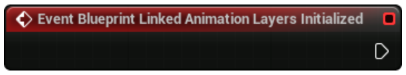
Event Graph

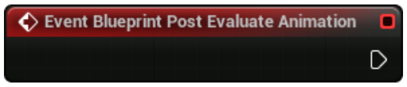
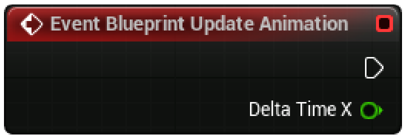
Every Animation Blueprint has a single **EventGraph**, which is a standard Graph that uses a collection of special animation-related events to initiate sequences of nodes. The most common use of the EventGraph is to update values or properties used by AnimGraph nodes.

Animation Events

The EventGraph is used by adding one or more events to act as entry points and then connecting Functions, Flow Control nodes, and Variables to perform the desired actions.

With the CPU Threading and Node Function features offered in the AnimGraph, it is recommended to use the EventGraph as little as possible. This is because the EventGraph executes on the primary game thread, along with most other Blueprint logic in your project. Therefore, having a complex EventGraph in your Animation Blueprint will reduce overall performance. Most of these events have thread-safe counterparts, and should be used instead if possible.

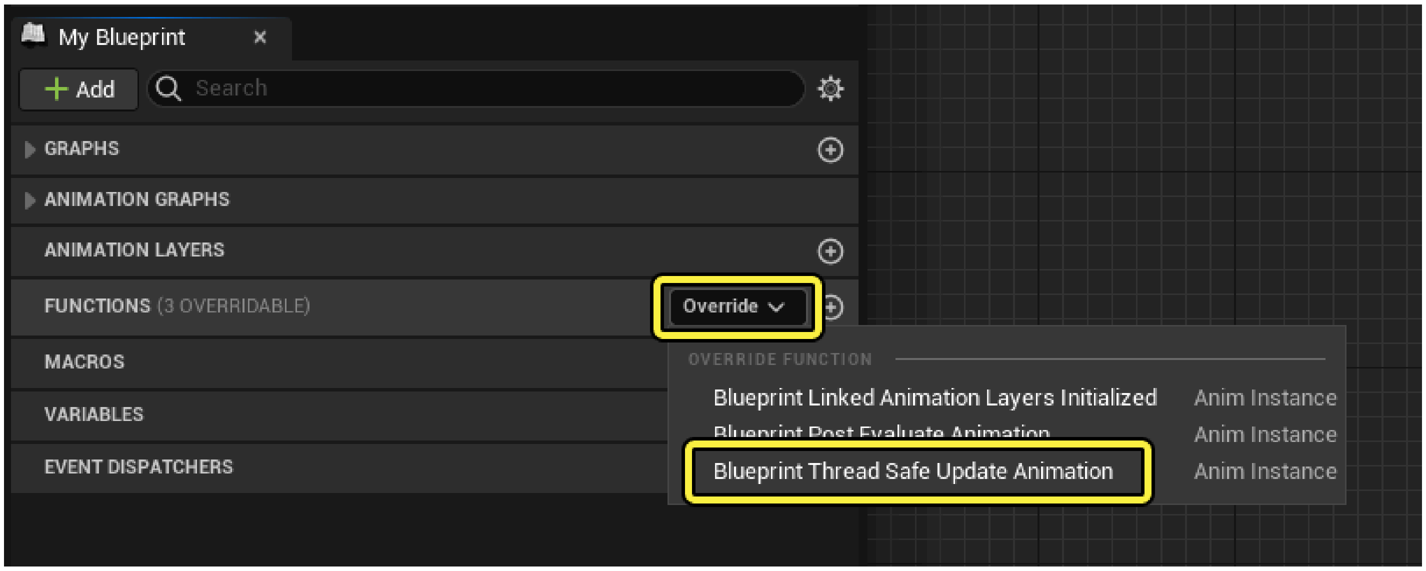
Event Name	Node image	Description
Begin Play	 Event Blueprint Begin Play	Similar to **Event Begin Play** in Blueprint Visual Scripting, this event is executed at the start of the game or simulation, but before the **Begin Play** event of an actor. As a thread-safe alternative, you can instead use the **On Initial Update** Node Function for the relevant node.
Initialize Animation	 Event Blueprint Initialize Animation	This event is executed once when the Animation Blueprint instance is created to perform initialization operations. It executes as soon as the Animation Blueprint is created, before the execution of an actor's **Construction Script** and **Begin Play**.
Linked Animation Layers Initialized	 Event Blueprint Linked Animation Layers Initialized	This event executes once, after Initialize Animation, and after any linked animation layers are initialized. As a thread-safe alternative, you can instead use the **On Initial Update** Node Function for the relevant linked animation layer node.

Event Name	Node image	Description
Post Evaluate Animation		Executes every frame, but after the animation finishes evaluating and has applied the pose for the current frame. This is useful to reset values, or to get the accurate transform of a bone.
Update Animation		Executes every frame, allowing the Animation Blueprint to perform calculations and updates to any values it needs. This event is the entry point into the update loop of the EventGraph. The time elapsed since the last update is available from the **Delta Time X** pin so time-dependent interpolations or incremental updates can be performed. As a thread-safe alternative, you can instead use the **Blueprint Thread Safe Update Animation Function**.
AnimNotify		Executes when a Skeleton Notify is triggered.

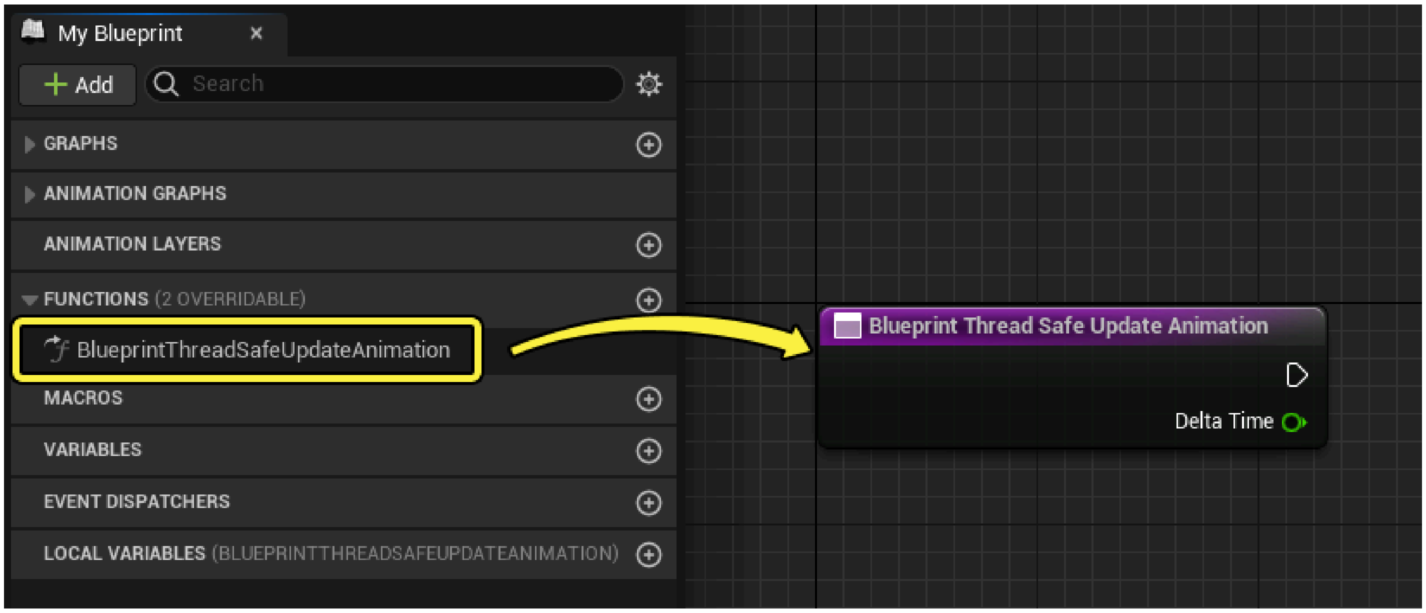
Thread Safe Update Animation

To improve the performance of your Animation Blueprint, you can use a thread-safe alternative to the Update Animation Event, called **Blueprint Thread Safe Update Animation**. This alternative is a Function that you must override in order to add it to the Blueprint. It is useful because the Event Graph Update Animation event always runs on the game thread, so it cannot take advantage of multithreading to improve overall framerate

To do this, click the **Override** dropdown menu in the **Functions** category of the **My Blueprint** panel, then select **Blueprint Thread Safe Update Animation.**



It is now added to your list of Functions. Opening it will reveal the function entry point, as well as a **Delta Time** pin, similar to the Delta Time X pin on the EventGraph Update Animation node. You can now create the same Update Animation logic in this Function as you would in the EventGraph, with this function executing on the Worker Thread instead of the Game Thread.


My Blueprint
Add
Search
GRAPHS
ANIMATION GRAPHS
ANIMATION LAYERS
FUNCTIONS (2 OVERRIDABLE)
BlueprintThreadSafeUpdateAnimation
MACROS
VARIABLES
EVENT DISPATCHERS
LOCAL VARIABLES (BLUEPRINTTHREADSAFEUPDATEANIMATION)
Blueprint Thread Safe Update Animation
Delta Time