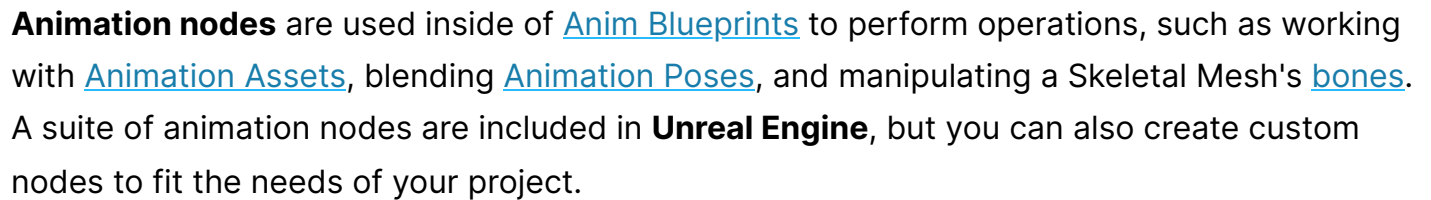
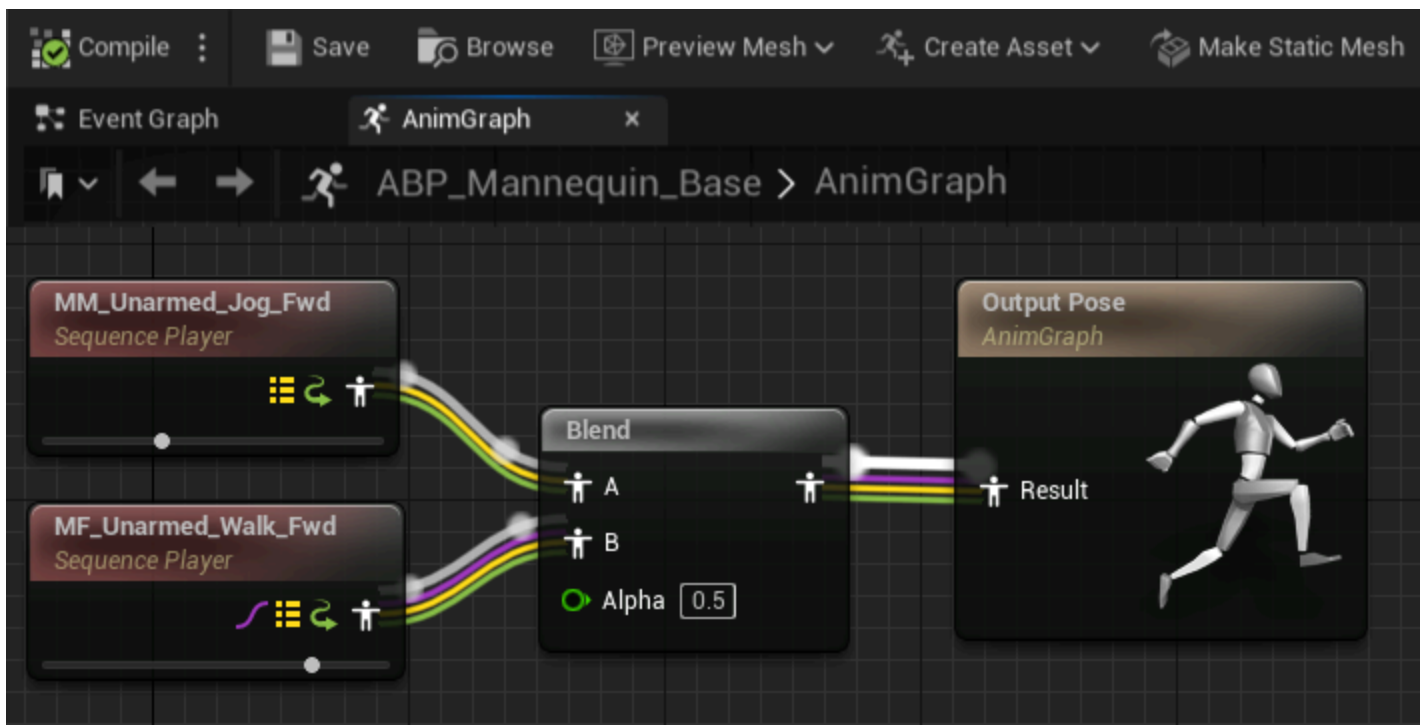


/ Animation Node Technical Guide

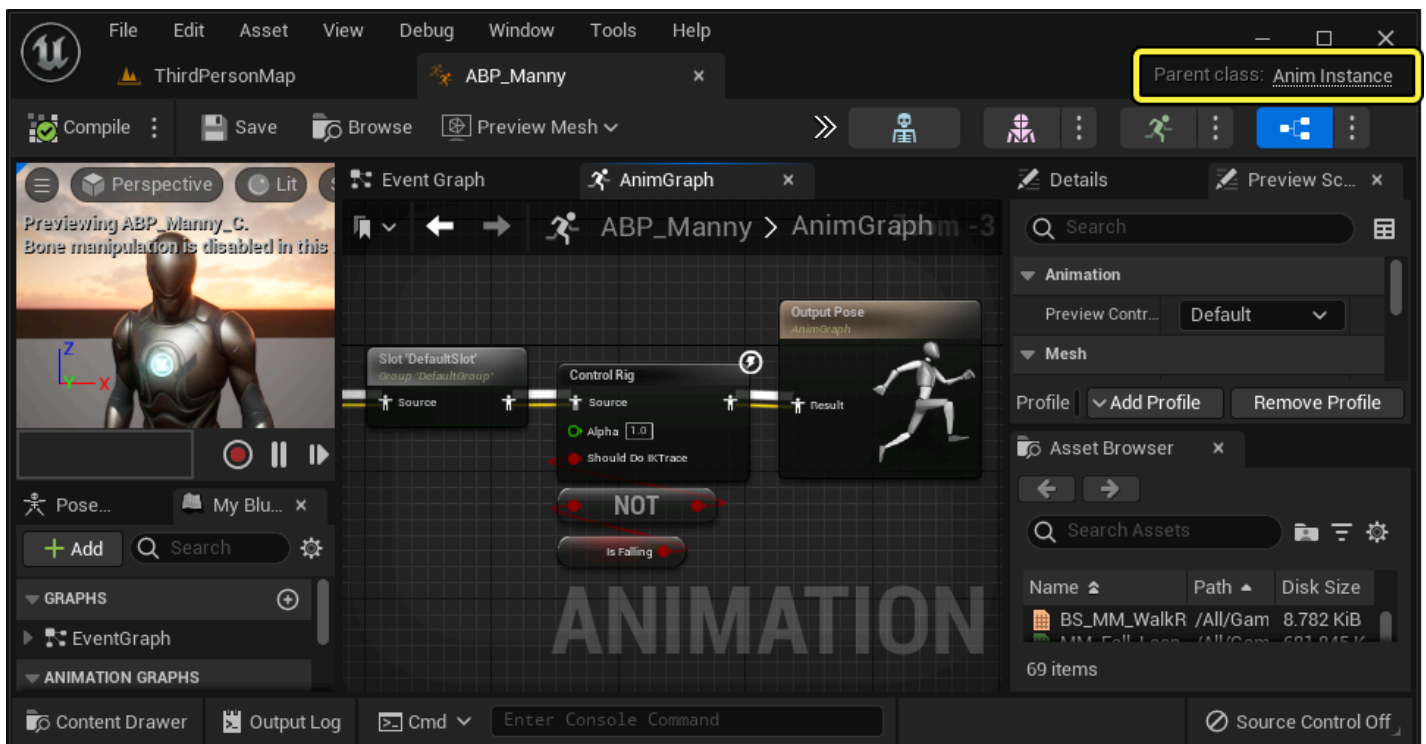
Guide to creating new nodes for use within graphs in Anim Blueprints.



Animation nodes are used inside of [Anim Blueprints](#) to perform operations, such as working with [Animation Assets](#), blending [Animation Poses](#), and manipulating a Skeletal Mesh's [bones](#). A suite of animation nodes are included in **Unreal Engine**, but you can also create custom nodes to fit the needs of your project.



To open an Animation Blueprint in your project's [source code editing IDE](#), open the AnimBP in the [AnimBP Editor](#), and click the **Parent Class** link in the top-left of the editor window.



Anatomy of an Animation Node

The two essential components of any Animation nodes are:

- A [runtime struct](#) that performs the actual operations to generate an output pose.

- An [editor-time container class](#) that handles visual aspects and functionality of the node within the graph, such as node titles and the context menu.

In order to add a new animation node, both of these must be created.

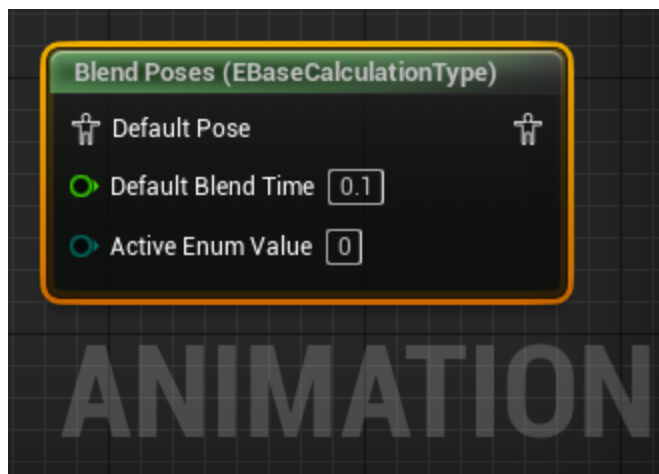
Node Hierarchies

It is possible to create a hierarchy of nodes, but any non-abstract editor-time classes should contain exactly one runtime node.



Do not add additional nodes when deriving unless the parent was abstract and did not contain one.

See the `UAnimGraphNode_BlendListBase` node family for examples.



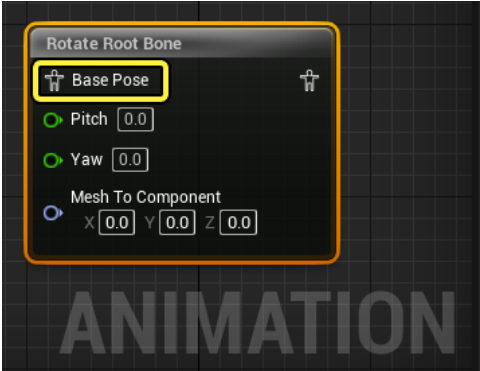
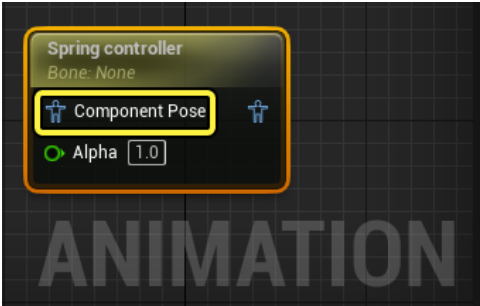
Runtime Node Component

The **runtime struct** is derived from `FAnimNode_Base` and is responsible for initialization, updating, and performing operations on one or more input poses to generate the desired output pose. It also declares any input pose links and any properties needed by the node to perform the desired operation.


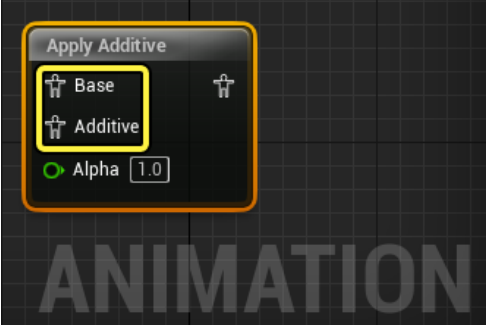
Pose Inputs

In the runtime node, pose inputs are exposed by creating properties of the type `FPoseLink` or `FComponentSpacePoseLink`. `FPoseLink` is used when working with poses in local space, such as blending animations. `FComponentSpacePoseLink` is used when working with poses in component space, such as applying skeletal controllers.

An Anim BP node can have a single pose input. The following are examples of animation nodes using a single pose input.

Node Classification	Code Example	Image
Local Space	<div><div>Local Space pose input code implementation:</div><div><div><div>1</div><div>UPROPERTY(Category=Links)</div></div><div><div>2</div><div>FPoseLink BasePose;</div></div></div><div><div>Copy full snippet</div></div></div>	<div></div> <div>Component Space pose input pins are blue.</div>
Component Space	<div><div>Component Space pose input code implementation:</div><div><div><div>1</div><div>UPROPERTY(Category=Links)</div></div><div><div>2</div><div>FComponentSpaceP oseLink ComponentPose;</div></div></div><div><div>Copy full snippet</div></div></div>	<div></div>

An Anim BP node can also have more than one pose input pin for nodes that blend between multiple animations:

Node Classification	Code Example	Image
Blending Node	<p>Base pose and Additive pose input pin code implementation:</p> <div><div>1</div><div>UPROPERTY(Category=Links)</div><div>2</div><div>FPoseLink Base;</div><div>3</div><div>4</div><div>UPROPERTY(Category=Links)</div><div>5</div><div>FPoseLink Additive;</div></div> <p> Copy full snippet</p>	

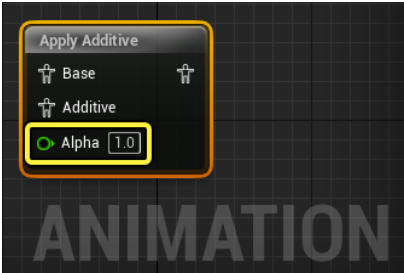
Once implemented into your custom Anim BP node, each of these properties will display a pose link input pin.



Properties of this type are always exposed as input pins. They cannot be optionally hidden or used only as editable properties in the **Details** panel.

Properties and Data Inputs

You can assign any number of properties to an AnimBP that are used to perform the operations of the node. Similar to other properties, you can declare custom properties using the `UPROPERTY` macro.

Node Clasifcation	Code Example	Image
Alpha Property Implementation	<div>Alpha property pin code implementation:</div> <div><div>1</div><div><code>UPROPERTY(Category=Settings, meta(PinShownByDefault))</code></div><div>2</div><div><code>mutable float Alpha;</code></div></div> <div><div></div>Copy full snippet</div>	

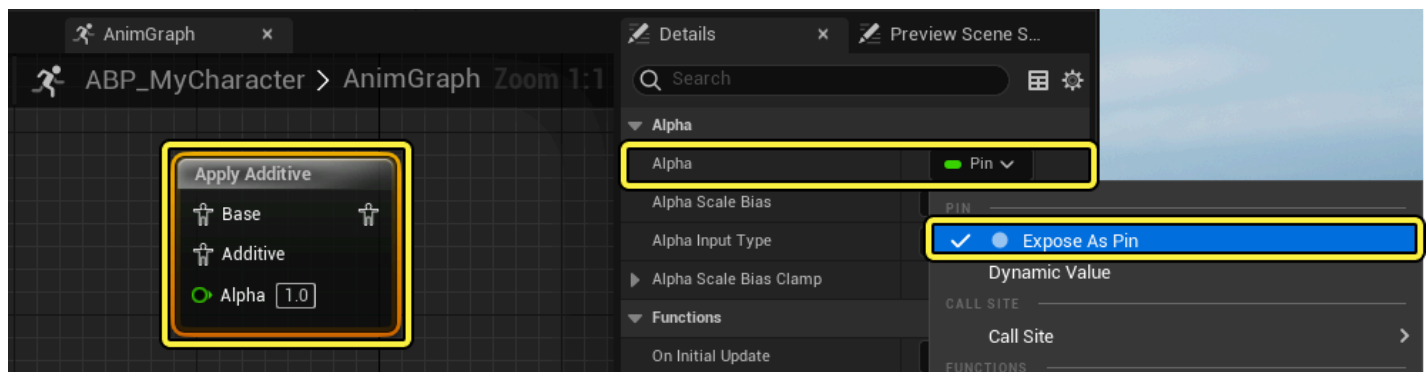
Using special **metadata keys**, animation node properties can be exposed as **data inputs pins** to allow values to be passed to the node. The following are metadata keys you can use when create custom AnimBP nodes for your project.

Metadata Key	Description
<code>NeverAsPin</code>	This key will hide property as a data pin in the AnimGraph and will only be editable in the node's Details panel.
<code>PinHiddenByDefault</code>	You can use this key to hide the property as a pin, by default. The property can then be exposed as a data pin in the AnimGraph. See the Optional Pins section for more information about exposing hidden pins in the AnimGraph.
<code>PinShownByDefault</code>	With this key you can expose a property as a data pin in the AnimGraph by default.

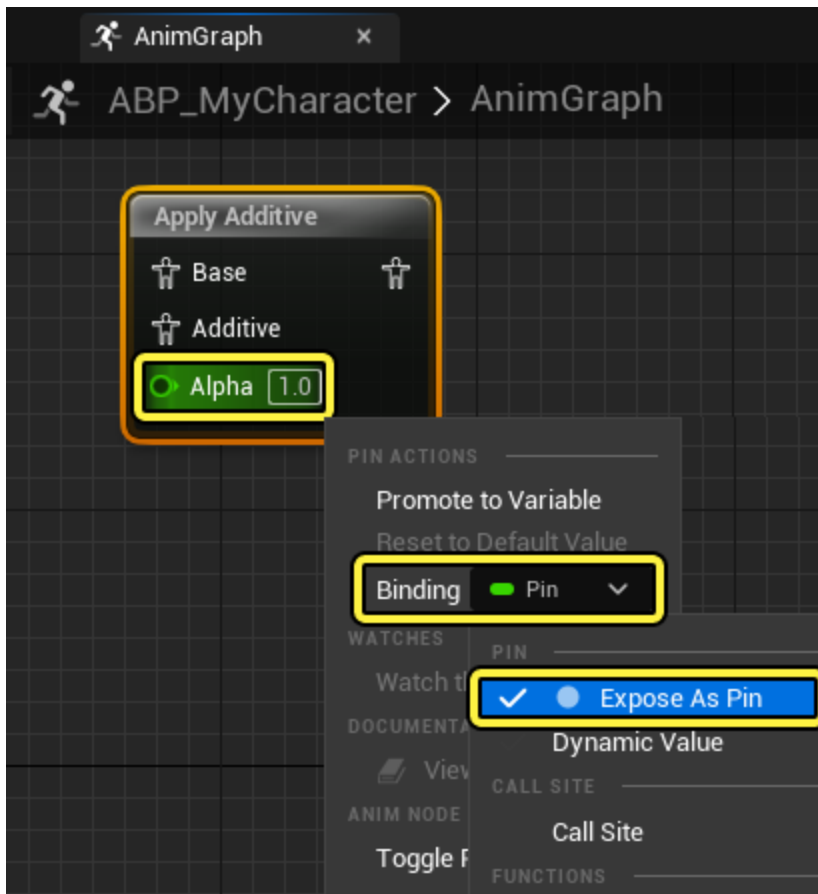
Metadata Key	Description
<code>AlwaysAsPin</code>	This key will always expose the property as a data point in the AnimGraph.

Optional Pins

For properties that are hidden, but exposable in the AnimGraph, using keys like `PinHiddenByDefault` or `PinShownByDefault`, you can expose properties in the node's **Details** panel, by navigating to the property and toggling **Expose As Pin** from the dropdown menu.



You can also hide property pins from the AnimGraph by **right-clicking** the pin you want to hide, navigating to the **Binding** option and toggling **Expose As Pin** from the dropdown menu.



Editor Node Component

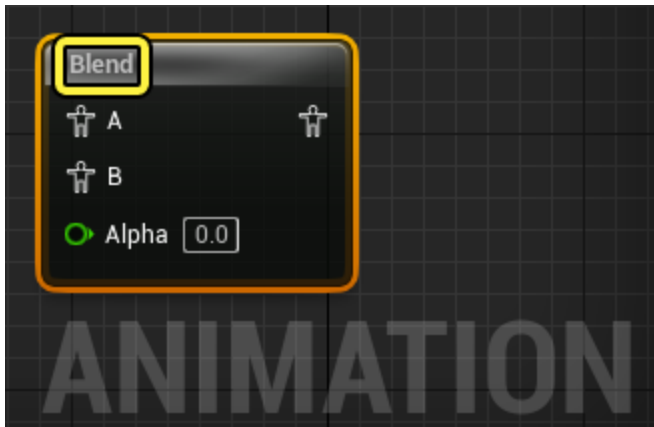
The editor class is derived from `UAnimGraphNode_Base` and is responsible for visual elements like the node's title or adding context menu actions.

The editor-time class should contain an instance of your runtime node exposed as editable.

```
1 UPROPERTY(Category=Settings)
2 FAnimNode_ApplyAdditive Node;
3
```

 Copy full snippet

Title



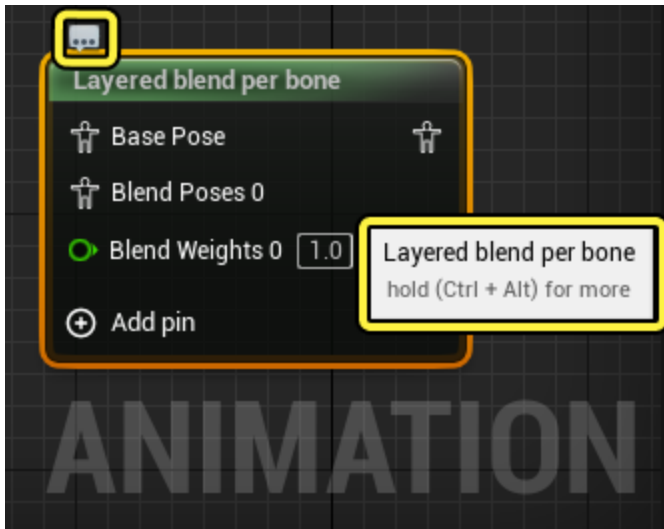
You can override the appearance of an Animation node's title elements in the AnimGraph, such as the text and background color, using the `GetNodeTitle` and `GetNodeTitleColor` functions.

For example, the `UAnimGraphNode_ApplyAdditive` node uses a gray background and displays "Apply Additive":

```
1 FLinearColor UAnimGraphNode_ApplyAdditive::GetNodeTitleColor() const
2 {
3     return FLinearColor(0.75f, 0.75f, 0.75f);
4 }
5
6 FString UAnimGraphNode_ApplyAdditive::GetNodeTitle(ENodeTitleType::Type
7     TitleType) const
8 {
9     return TEXT("Apply Additive");
10 }
```

 Copy full snippet

Tooltip



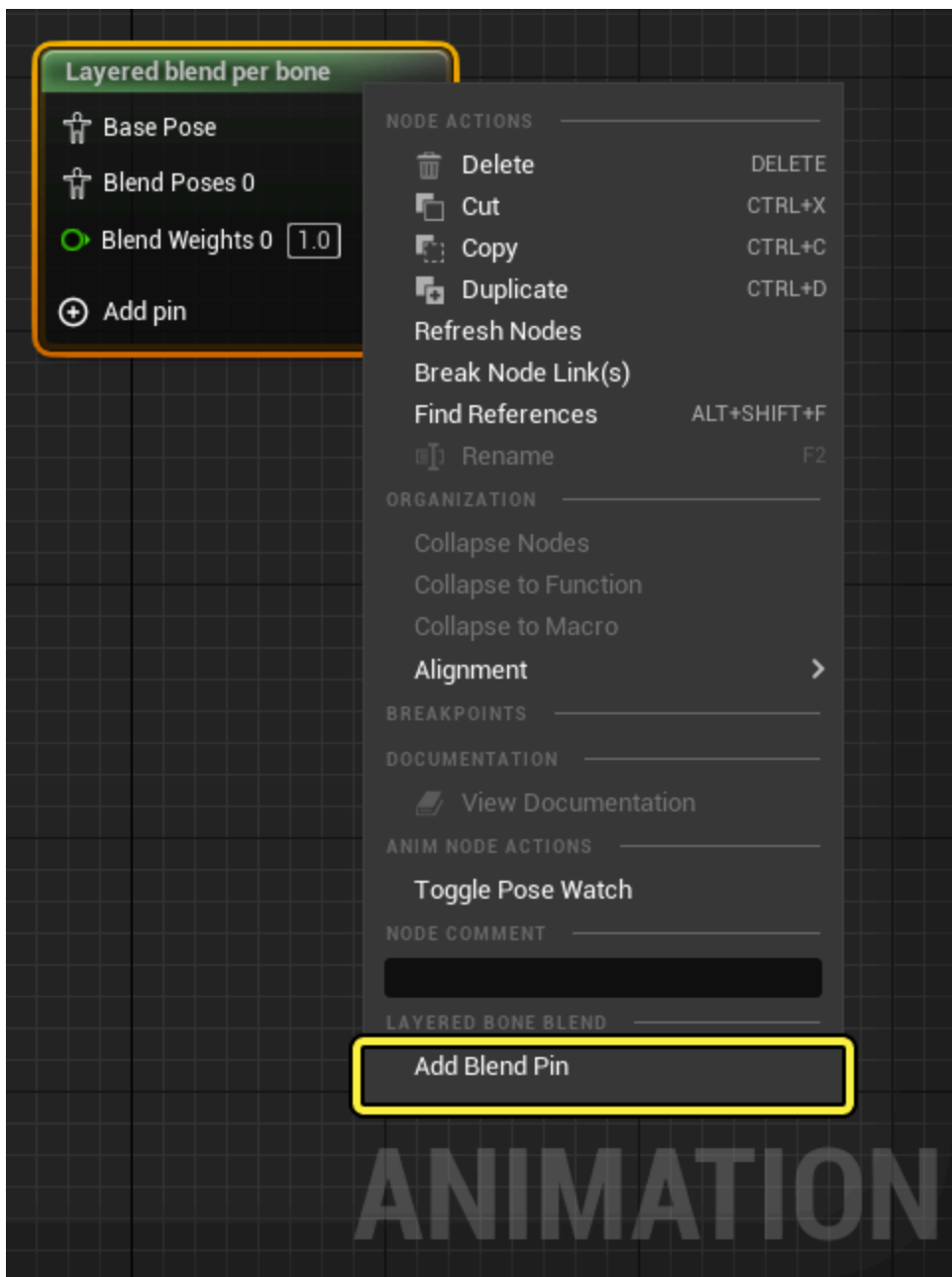
When creating custom animation nodes you can create custom tooltips that are viewable in the AnimGraph by overriding the `GetTooltip` function:

```
1 FString UAnimGraphNode_ApplyAdditive::GetTooltip const
2 {
3     return TEXT("Apply additive animation to normal pose");
4 }
5
```

 Copy full snippet

Context Menu

When creating your own custom animation nodes, you can add node-specific options to the node's context menu, which is accessible by **right-clicking** the node in the AnimGraph. You can add context menu options to your custom animation nodes using the `GetContextMenuActions` function, which also is a function of all Blueprint nodes in Unreal Engine.



For example, the `UAnimGraphNode_LayeredBoneBlend` node adds context-menu options for adding a **Add Blend Pin** or **Remove Blend Pin**:

```

1 void UAnimGraphNode_LayeredBoneBlend::GetContextMenuActions(const
  FGraphNodeContextMenuBuilder& Context) const
2 {
3   if (!Context.bIsDebugging)
4   {
5     if (Context.Pin != NULL)
6     {
7       // we only do this for normal BlendList/BlendList by enum, BlendList by Bool
        doesn't support add/remove pins
8     if (Context.Pin->Direction == EGPD_Input)

```

```

9 {
10 //@TODO: Only offer this option on arrayed pins
11 Context.MenuBuilder->BeginSection("AnimNodesLayeredBoneBlend",
    NSLOCTEXT("A3Nodes", "LayeredBoneBlend", "Layered Bone Blend"));
12 {
13 Context.MenuBuilder-
    >AddMenuEntry(FGraphEditorCommands::Get().RemoveBlendListPin);
14 }
15 Context.MenuBuilder->EndSection();
16 }
17 }
18 else
19 {
20 Context.MenuBuilder->BeginSection("AnimNodesLayeredBoneBlend",
    NSLOCTEXT("A3Nodes", "LayeredBoneBlend", "Layered Bone Blend"));
21 {
22 Context.MenuBuilder-
    >AddMenuEntry(FGraphEditorCommands::Get().AddBlendListPin);
23 }
24 Context.MenuBuilder->EndSection();
25 }
26 }
27 }
28

```

 Copy full snippet

Derived Native Getters

You can create your own `UAnimInstance` derived class to achieve performance improvements. You can add new getters if there is a need for improved performance. You can set up a new getter following the steps below:

- The getter functions must be tagged as **UFUNCTIONS**.
- They must be **BlueprintPure**.
- They must include the metadata **AnimGetter="True"**.

They must also define some specifically named parameters (this is also explained above the base anim getter functions in `AnimInstance.h`). That list of parameters includes :

Parameter	Description
int32 AssetPlayerIndex	The getter acts on an asset player and an entry will be added to the editor per asset player available.
int32 MachineIndex	The getter acts on a state machine, an entry will be added per state machine.
int32 StateIndex	This also requires MachineIndex. The getter acts on a state, an entry will be added per state.
int32 TransitionIndex	This also requires MachineIndex. The getter acts on a transition, an entry will be added per transition.

You can also use helper functions to get the actual nodes in your getters. These exist on the

`UAnimInstance`:

Function	Description
GetStateMachineInstance(int32 MachineIndex)	Gets the baked state machine instance.
GetCheckedNodeFromIndex(int32 NodeIdx)	Gets a node from an index, asserts if invalid.
GetNodeFromIndex(int32 NodeIdx)	As above, can return nullptr.
GetRelevantAssetPlayerFromState(int32 MachineIndex, int32 StateIndex)	Gets the highest weighted asset player in a state.