Contents

About Rukhanka

Rukhanka is an animation system for Entity Component System (ECS) for Unity Technology Stack. It depends on `Unity Entities` and `Unity Entities Graphics` packages.

Design and implementation of **Rukhanka** follows three principles:

- Trivial usage and interface
- Performance in all aspects
- Functionality and behavior are identical to `Unity Mecanim Animation System`

Simple Interface

Rukhanka has a very limited set of own user interfaces. It has no complex custom editor windows and configurable options. Everything related to animation functionality is set up using familiar Unity editors. At bake time, **Rukhanka** converts standard Unity `Animators`, `Animation Clips`, and `Skinned Mesh Renderers` into their internal structures and works with them in runtime.

Performance

Everything in **Rukhanka** is designed with performance in mind. All core systems are `ISystem` based and `Burst` compiled. Core animation calculation and state machine processing loops fully benefit from multi-core/multi-processor systems. Even debug and visualization functionality, despite that it can be completely compiled out, made `Burst` compatible as much as possible.

'Mecanim'-like behaviour

Rukhanka tries to mimic the behavior of the `Unity Mecanim Animation System`. It tries to do this during state machine processing as well as animation calculation and blending. Some parts of `Mecanim` have not been implemented yet/made similar by 100% in **Rukhanka**. Refer to *feature summary tables* for detailed information on compatibility and support features.

Links

- This documentation: https://docs.rukhanka.com
- Youtube channel: https://www.youtube.com/@rukhankaanimation
- Discord Support Server: https://discord.gg/AwzFjWdHfq
- Support e-mail: support@rukhanka.com

Getting Started

Prerequisites

To work with **Rukhanka Animation System** you need following:

- Unity 2022.2.0f1+
- Unity `Entities` package version 1.0.0-pre.15 (installed automatically as dependency)
- Unity `Entities.Graphics` package version 1.0.0-pre.15 (installed automatically as dependency)
- HDRP or URP as required by `Entities.Graphics` package

Important: `Humanoid` animation types not supported yet. Only `Generic` animation types can be used at this moment.

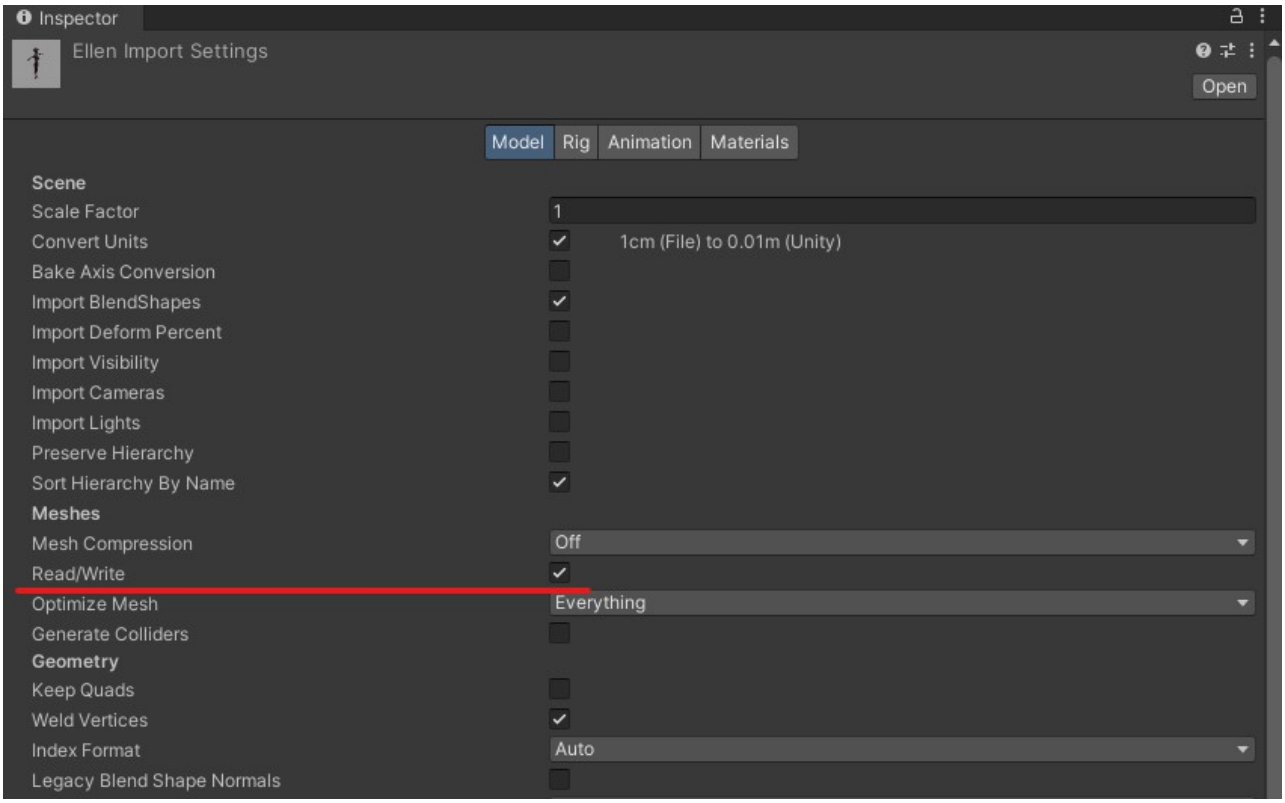
Animated Object Setup

To make animations work correctly there are some preparation setup steps are required.

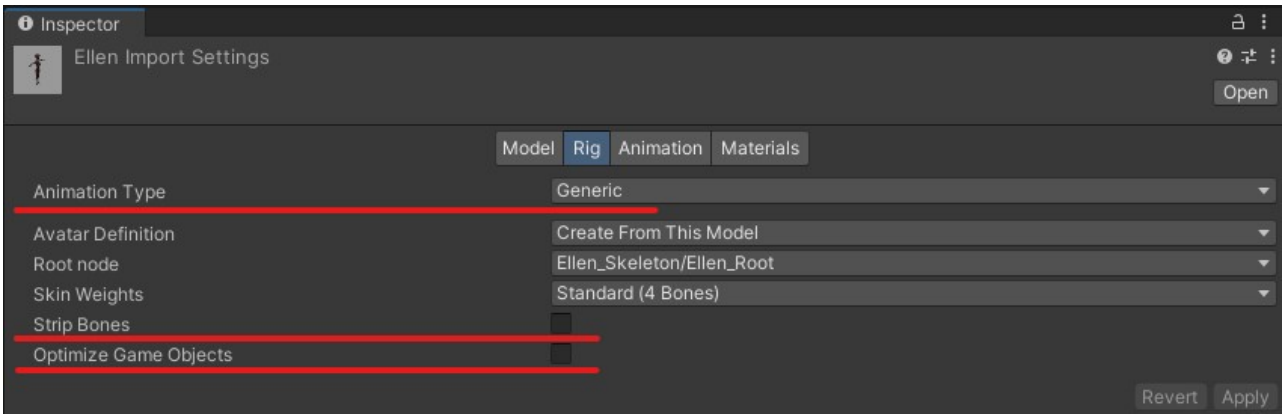
Model Importer

Use standard Unity model importer confuguration page to setup required model properties:

1. Model Importer Tab.
    - Enable `Read/Write` property for the mesh. This is requirement of `Deformation Subsystem` of `Entities.Graphics` package.



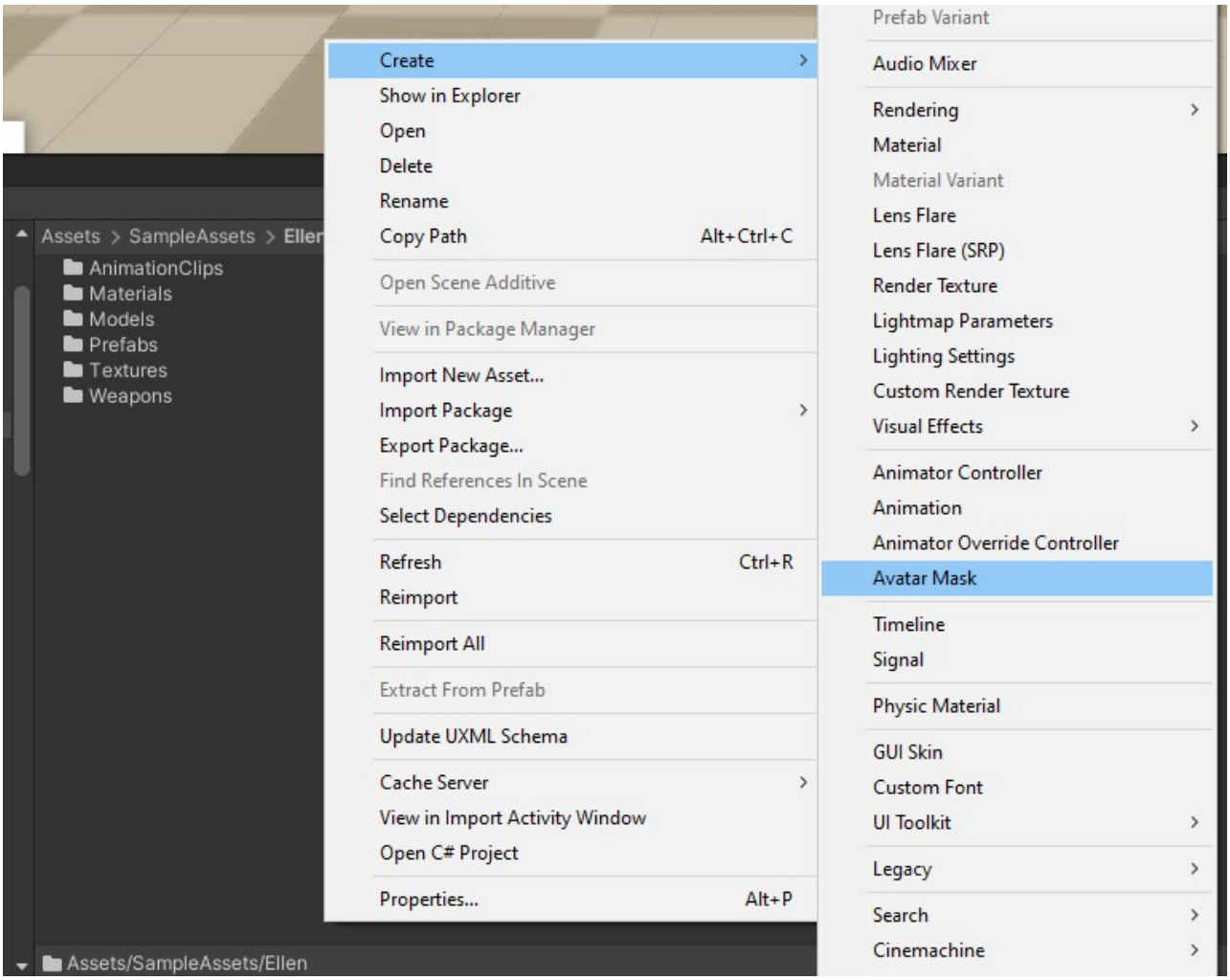
2. Rig Importer Tab.
    - Set `Animation Type` to `Generic`. Other types (particulary `Humanoid`) not supported yet.
    - Uncheck `Strip Bones` checkbox. **Rukhanka** need full unmodified hierarchy for own avatar setup.
    - Uncheck `Optimize Game Objects` checkbox. **Rukhanka** need all bone game objects in baking phase.



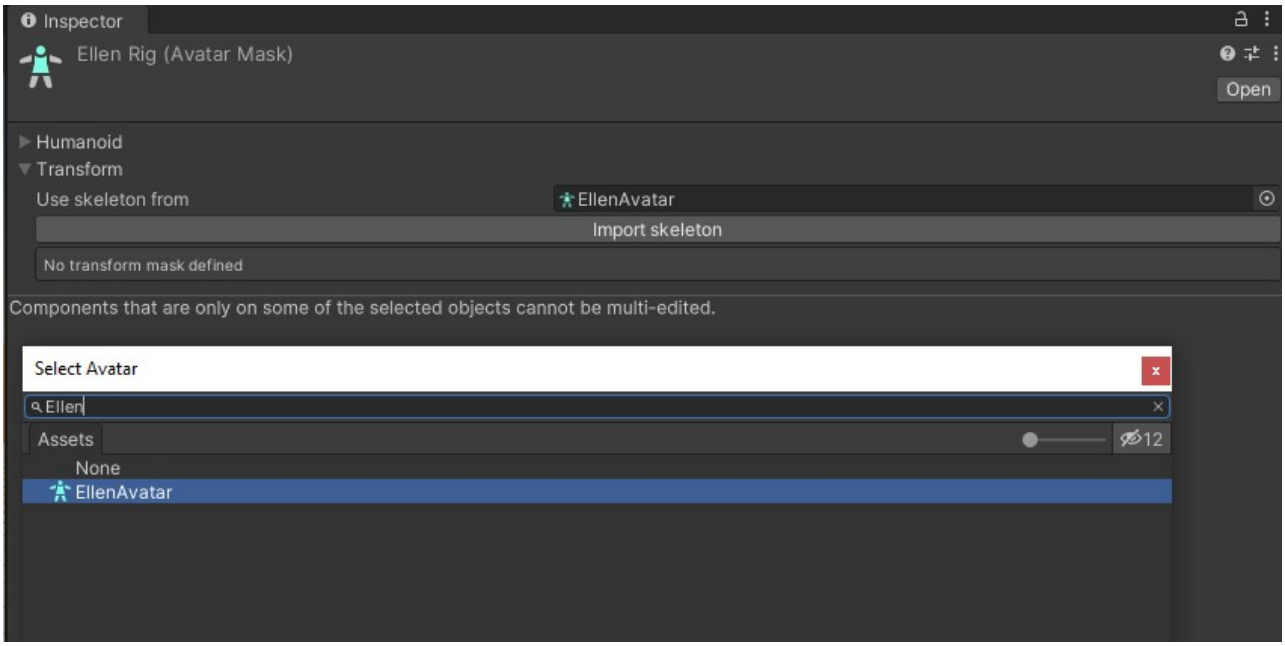
Rig Definition

Rukhanka baking systems cannot use `Unity Avatar` because of lack of required public interfaces in it. It is necessary to create own `Rig Definition` from `Unity Avatar`. **Rukhanka** uses internal Unity's `Avatar Mask` functionality to define Rig bones hierarchy. All transform data will be gathered from `GameObject` hierarchy during baking.

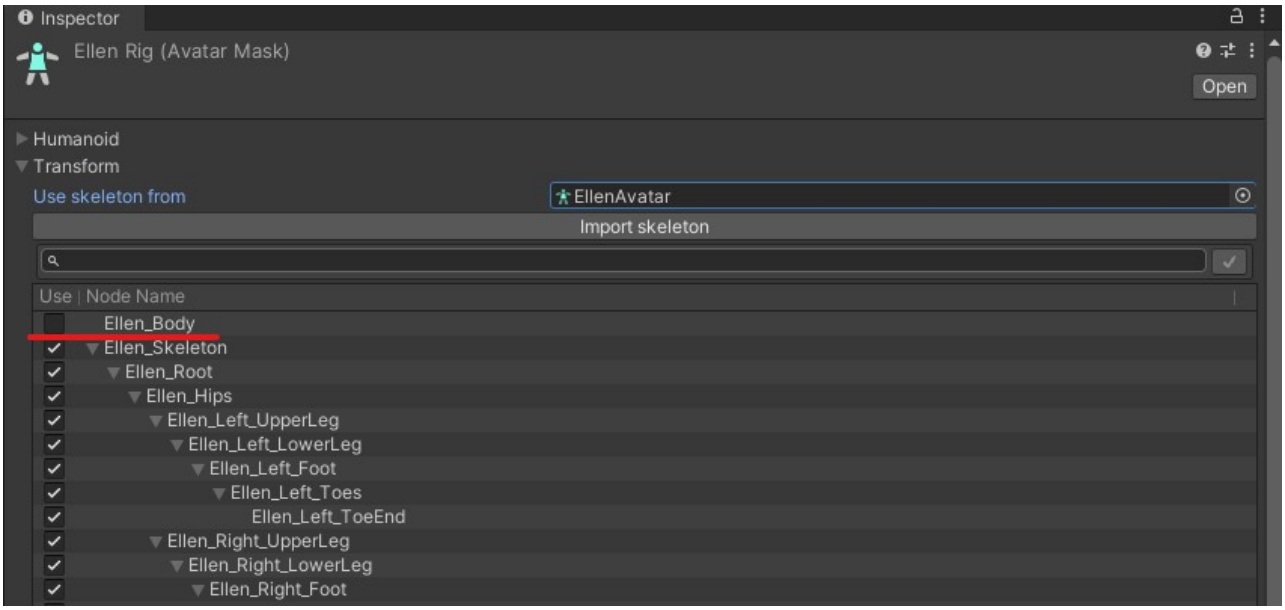
1. Create new `Avatar Mask` somwhere inside project assets directories.



2. In `Avatar Mask` inspector pick your model `Avatar` in `Transform->Use skeleton from` field.



3. Press `Import skeleton` button to import entire avatar rig hierarchy into this `Avatar Mask`.
4. Expand all nodes and make sure that every bone is selected and meshes are not selected.

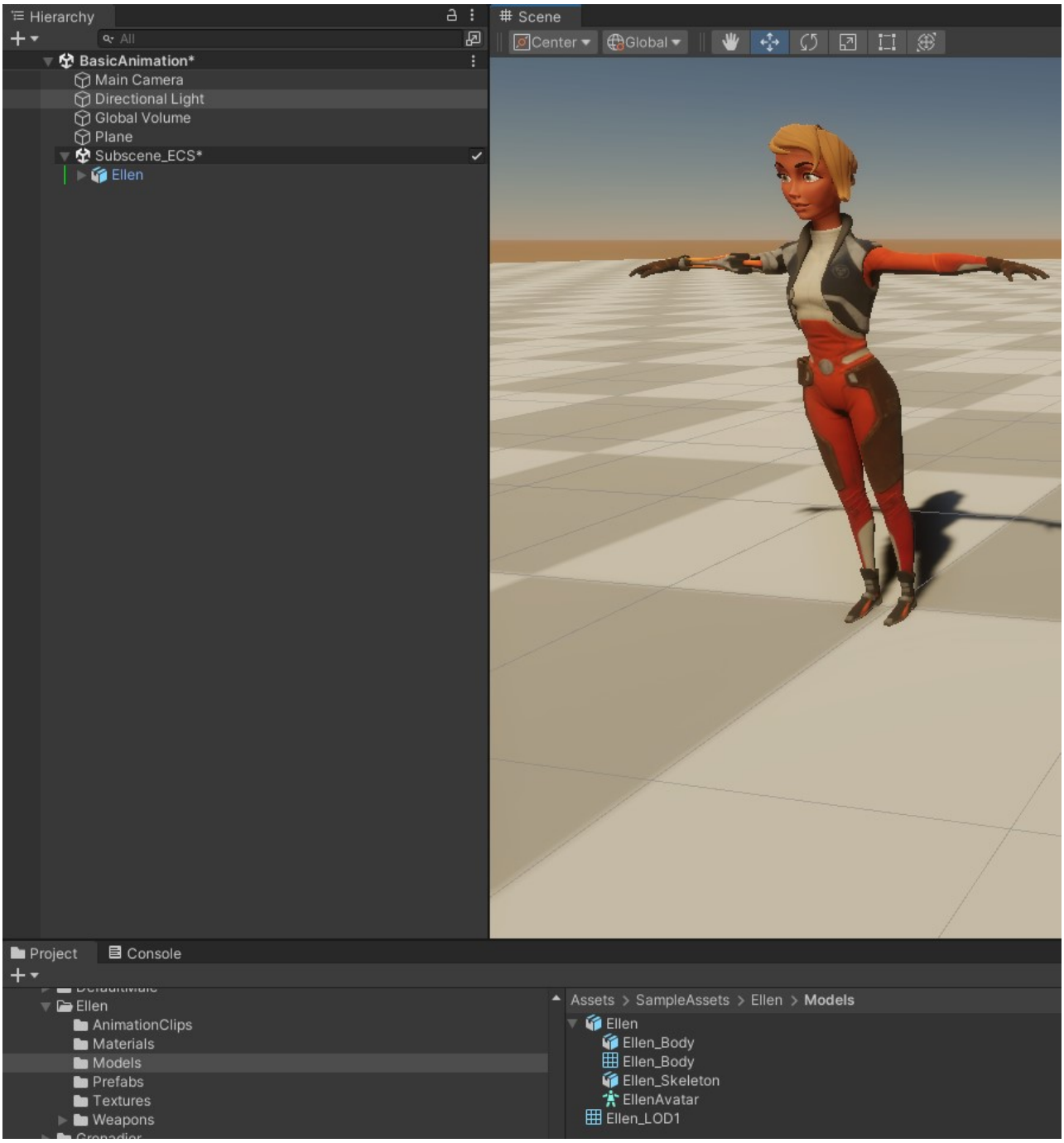


5. Do not forget to rename your new `Avatar Mask` to something meaningful (like `EllenRig` in this examples) to indicate that this is not ordinary `Avatar Mask` but **Rukhanka** `Rig Definition`.

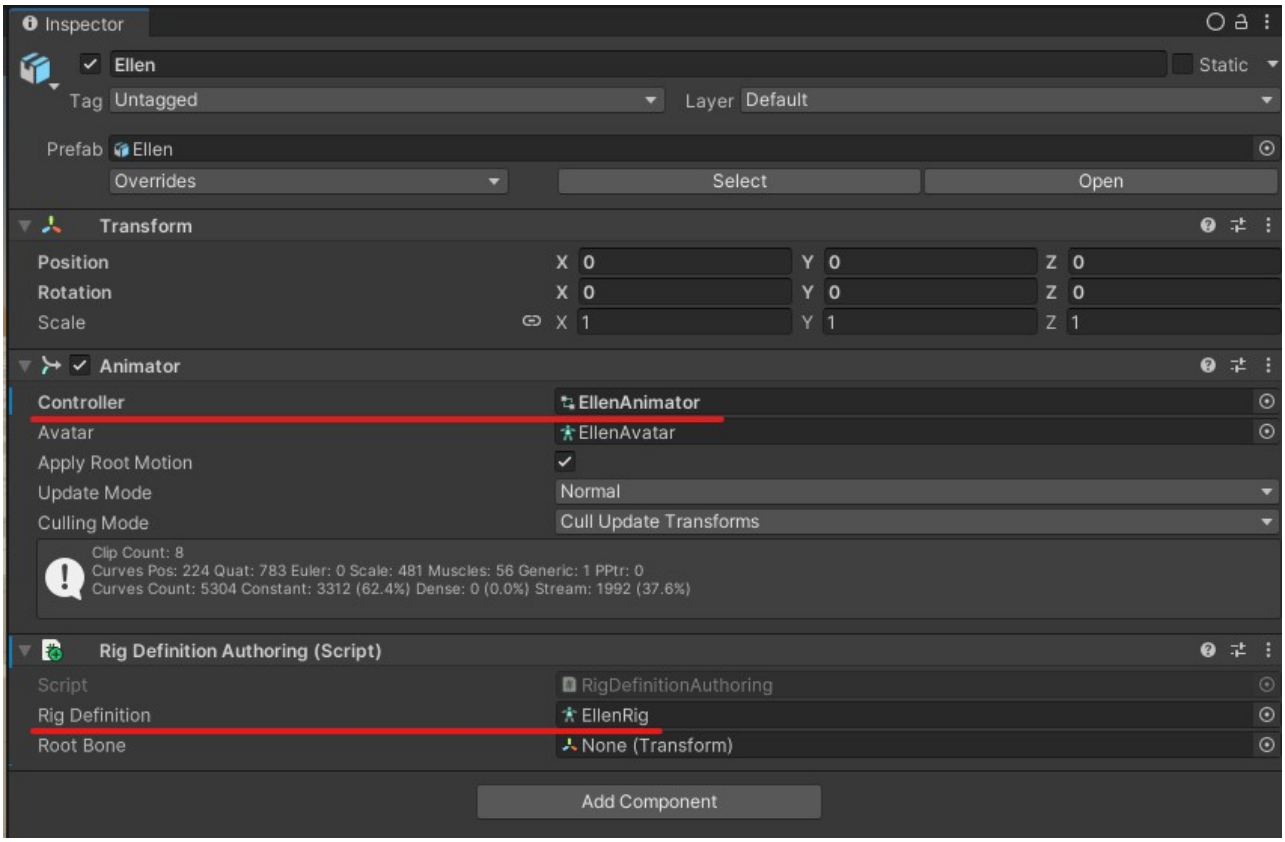
Authoring Object Setup

Final step is to create authoring `GameObject` inside `Entities` Subscene

1. Place your animated object inside `Entities` Subscene. For detailed description of this step refer Entites Package documentation.

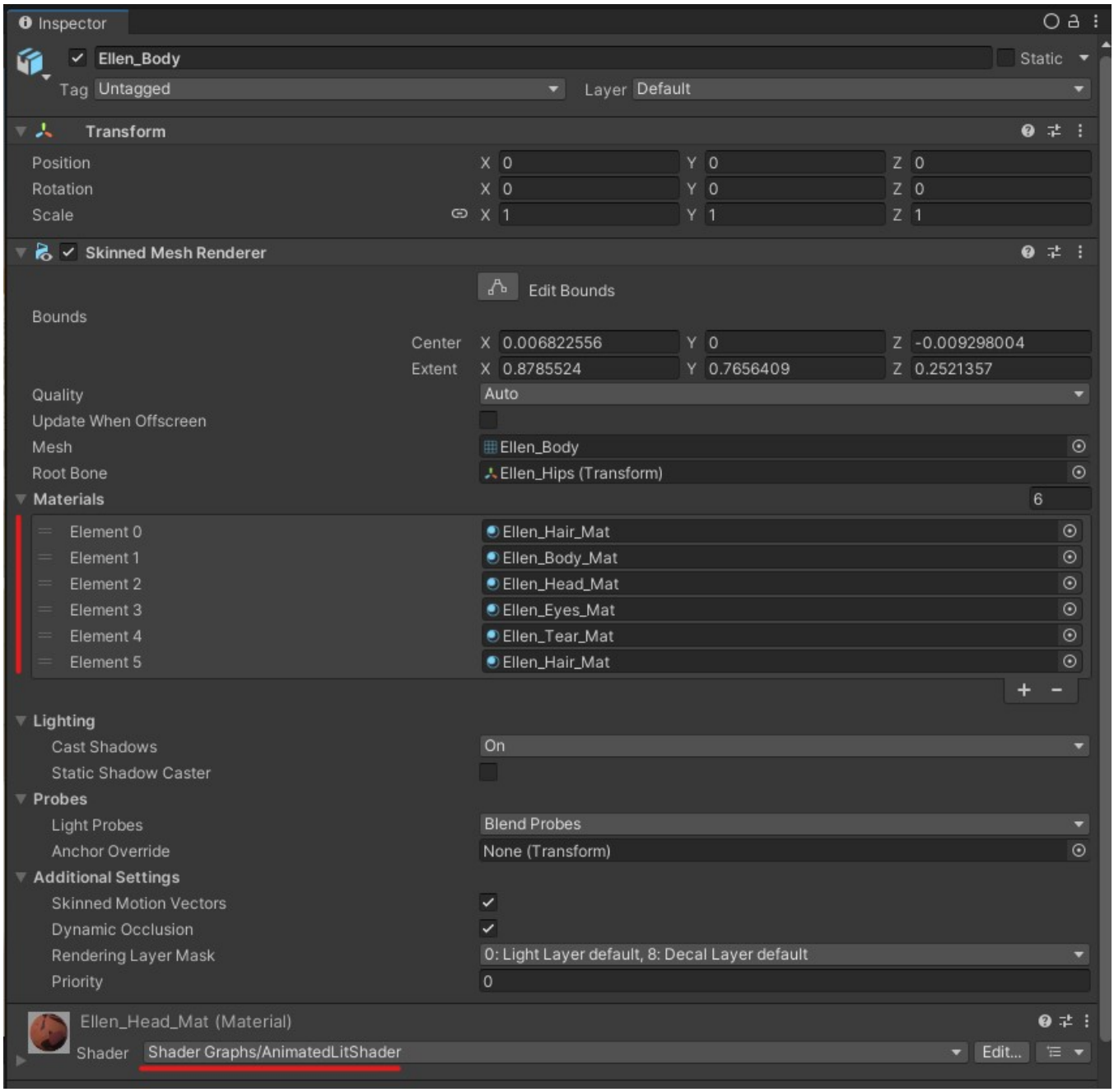


2. Add `Rig Definition Authoring` component to newly created object.
3. Place `Rig Definition` created previously in `Rig Definition` slot of `Rig Definition Authoring` component.
4. Create standard `Animator Controller` and fill it as you wish (one state with one animation will be good start).



Shaders And Materials

Rukhanka does not renders animated objects. It only prepares skin matrices for skinned meshes that are entirely managed by `Entities.Graphics` package. To be able to render deformed meshes correctly it is required to make `Entities.Graphics` compatible deformation-aware shader. Read carefully Mesh deformations section of `Entities.Graphics` package documentation. Make compatible shader. Make and assign all required materials on your animated model.



End Of Setup Process

That's all needed to make **Rukhanka** be able to convert `Animator Controller`, all required `Animations` and own `Rig Definition` into internal structures. After that runtime systems will simulate state machine behaviour and play requred animations.

IMPORTANT: There is not 100% Unity's `Mecanim` feature support. Please consult Feature Support Tables for complete information.

Here is video version of the entire *Getting Started* process

Animator Parameters

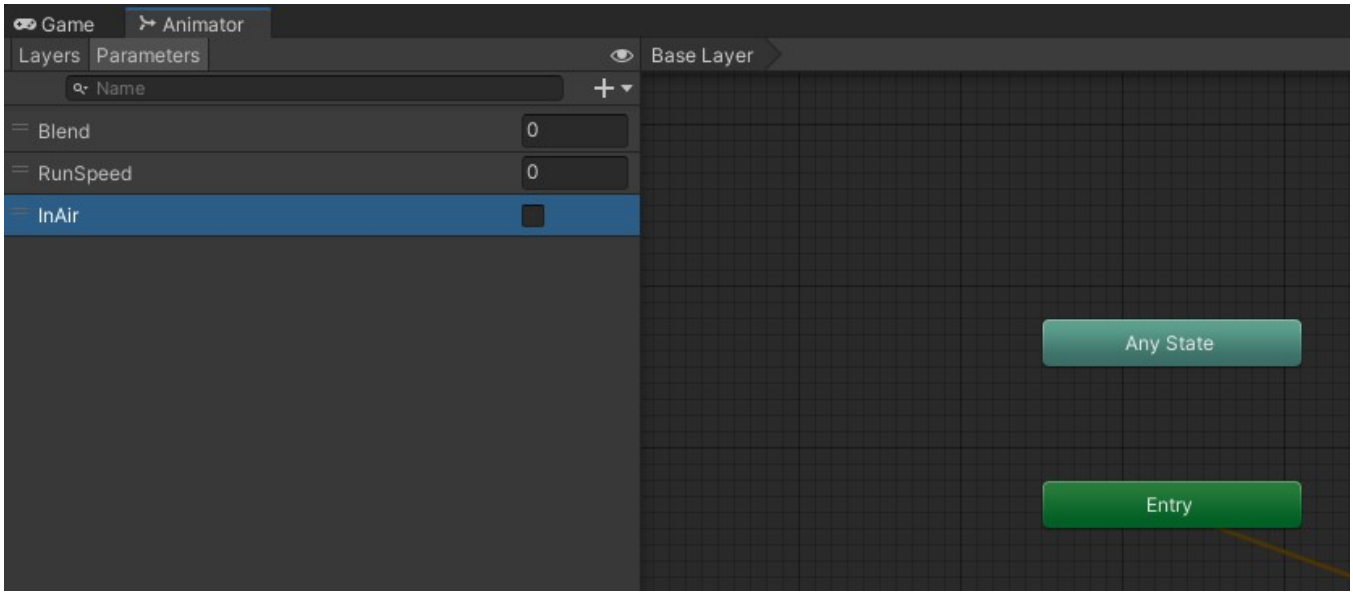
Direct Buffer Indexing

Rukhanka made `DynamicBuffer` for all animator parameters so the user can control its values from the code.

Basically, only `DynamicBuffer` with animator parameters is needed to access and manipulate parameter values. This is shown in the following code snippet:

```
[BurstCompile]
partial struct ProcessInputJob: IJobEntity
{
    void Execute(ref DynamicBuffer<AnimatorControllerParameterComponent> allParams)
    {
        var someParameter = allParams[0];
        //  Increment parameter
        someParameter.FloatValue += 1.0f;
        //  Put value back in the array
        allParams[0] = someParameter;
        ...
    }
}
```

This approach has only one advantage: access speed. Animator parameters are ordered in a way how they are defined in Unity's `Animator` from top to bottom. For example, in `Animator` parameters given in the next picture, there are three parameters: [0] - "Blend", [1] - "Run Speed", [2] - "InAir":



Perfect Hash Table

Accessing parameters by index has no name-value relationship. Any animator parameter reordering in `Animator Controller` will break the game logic code. So there is a better solution: accessing using a hash table. **Rukhanka** prepares `Perfect Hash Table` for a list of parameters during the baking stage. A perfect hash table is a hash table that has an unambiguous 'parameter name' -> 'array index' relationship. It is faster than ordinary hash tables and also has O(1) access complexity.

To simplify access to the parameters using a hash table, the helper class named `FastAnimatorParameter` is introduced. Follow these steps to access the animator parameter by name and in a very performant way:

- Define required `FastAnimatorParameters` as, for example, system private fields:

```
public partial class PlayerControllerSystem: SystemBase
{
    FastAnimatorParameter blendParam  = new FastAnimatorParameter("Blend");
    FastAnimatorParameter runSpeedParam  = new FastAnimatorParameter("RunSpeed");
```

```
    FastAnimatorParameter inAirParam  = new FastAnimatorParameter("InAir");
    ...
}
```

- Pass prepared `FastAnimatorParameters` in the job:

```
protected override void OnUpdate()
{
    var processInputJob = new ProcessInputJob()
    {
        blendParam = this.blendParam,
        runSpeedParam = this.runSpeedParam,
        inAirParam = this.inAirParam
    };

    ...
}
```

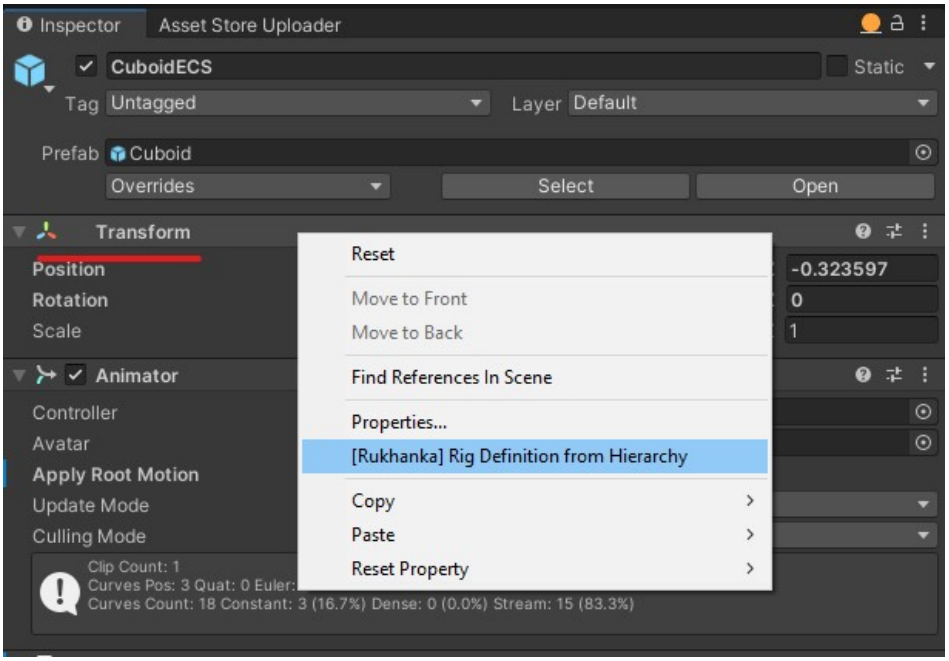
- Query `AnimatorControllerParameterIndexTableComponent` component (it is the perfect hash table for a set of animator parameters) and use the `FastAnimatorParameter` methods to access parameter value:

```
[BurstCompile]
partial struct ProcessInputJob: IJobEntity
{
    public FastAnimatorParameter blendParam;
    public FastAnimatorParameter runSpeedParam;
    public FastAnimatorParameter inAirParam;

    void Execute(in AnimatorControllerParameterIndexTableComponent paramIndexTable, ref DynamicBuf
    {
        var t = paramIndexTable.seedTable;

        blendParam.GetRuntimeParameterData(t, allParams, out var blendParamvalue);
        blendParam.SetRuntimeParameterData(t, allParams, new ParameterValue() { floatValue = blend
        inAirParam.SetRuntimeParameterData(t, allParams, new ParameterValue() { boolValue = true }
    }
}
```

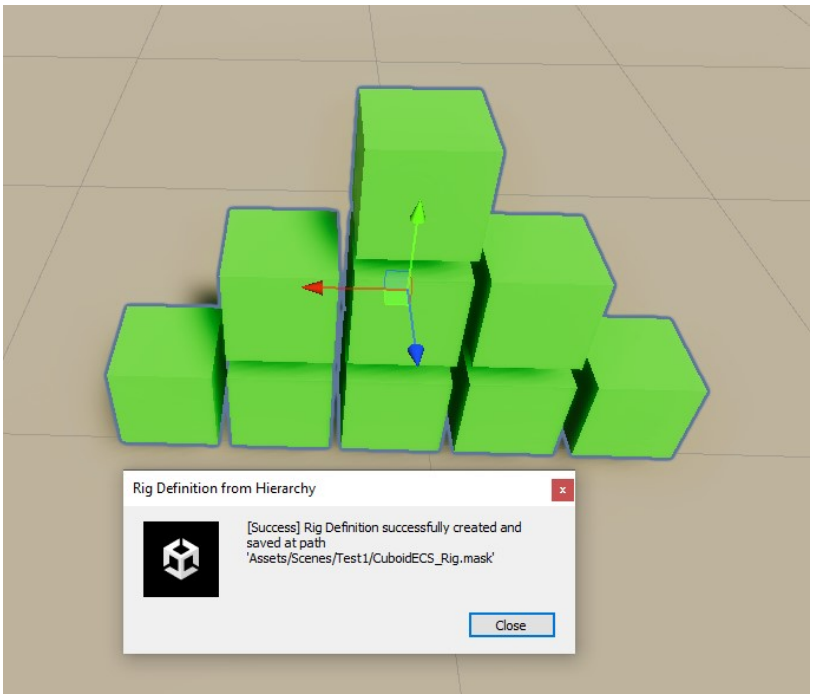
Non-skinned Mesh Animation

Rukhanka can animate arbitrary Entity hierarchy. The process is essentially the same as for ordinary skinned meshes:

- Create an `Animator` and define animations in it.
- Place an object on subscene and add the `Rig Definition Authoring` Unity component to it.
- Use special **Rukhanka**'s function to make `Rig Definition` from the `GameObject` hierarchy. Right-click on the hierarchy root `Transform` component caption, and choose the `[Rukhanka] Rig Definition from Hierarchy` option:



- Rig Definition `Avatar Mask` asset will be created in the current project directory:

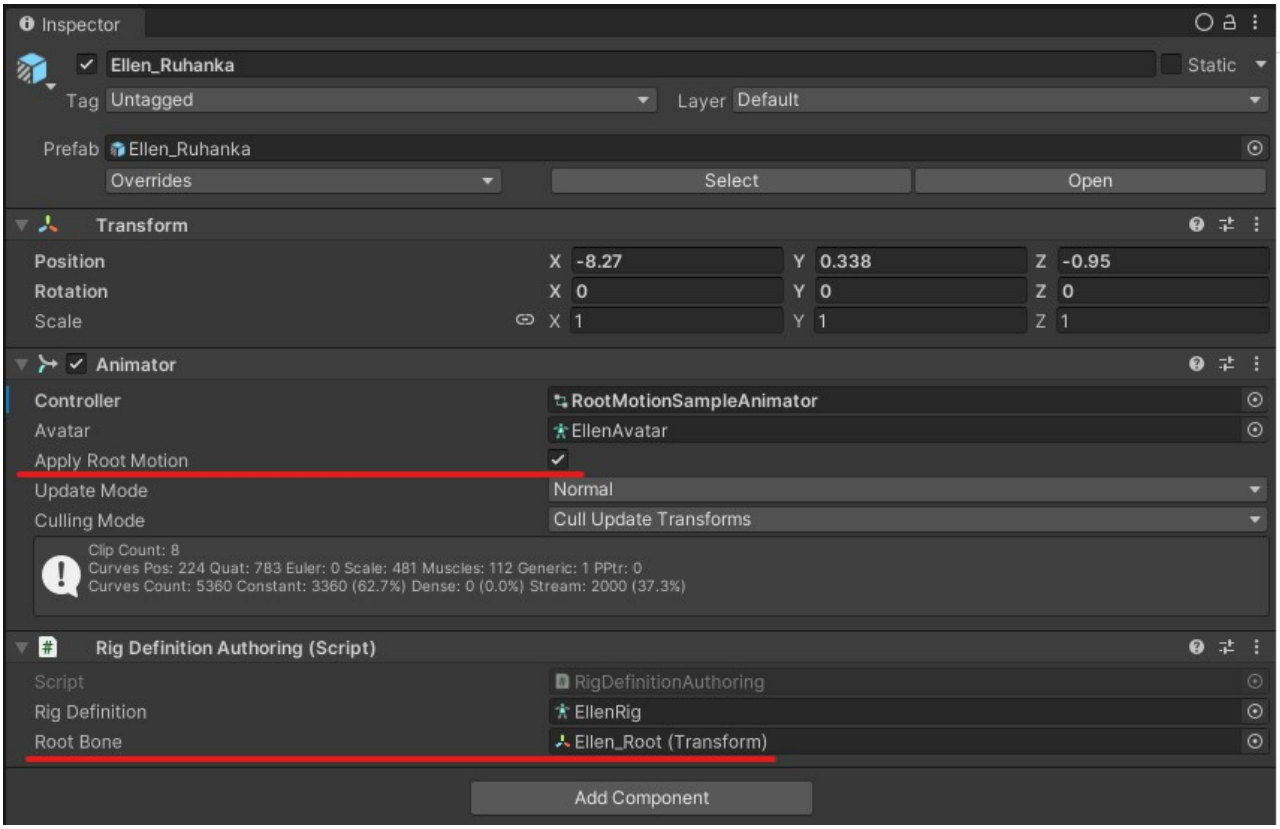


- Assign newly created `Rig Definition` into the `Rig Definition` field of the `Rig Definition Authoring` component.
- Add Unity `Animator` component, and assign the `Animator` state machine to it.

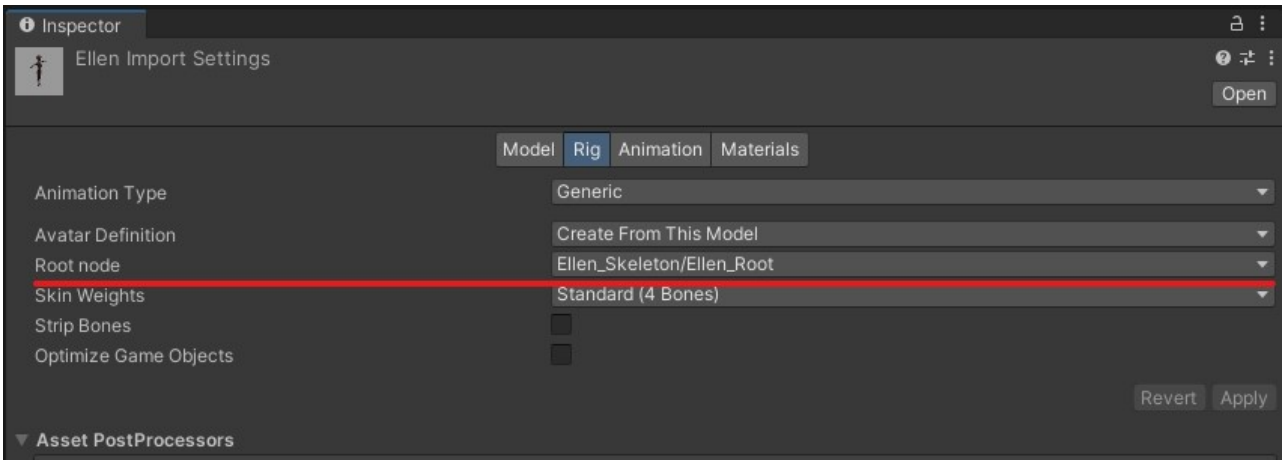
Root Motion

Rukhanka has limited root motion support. Follow these steps to enable root motion for your model:

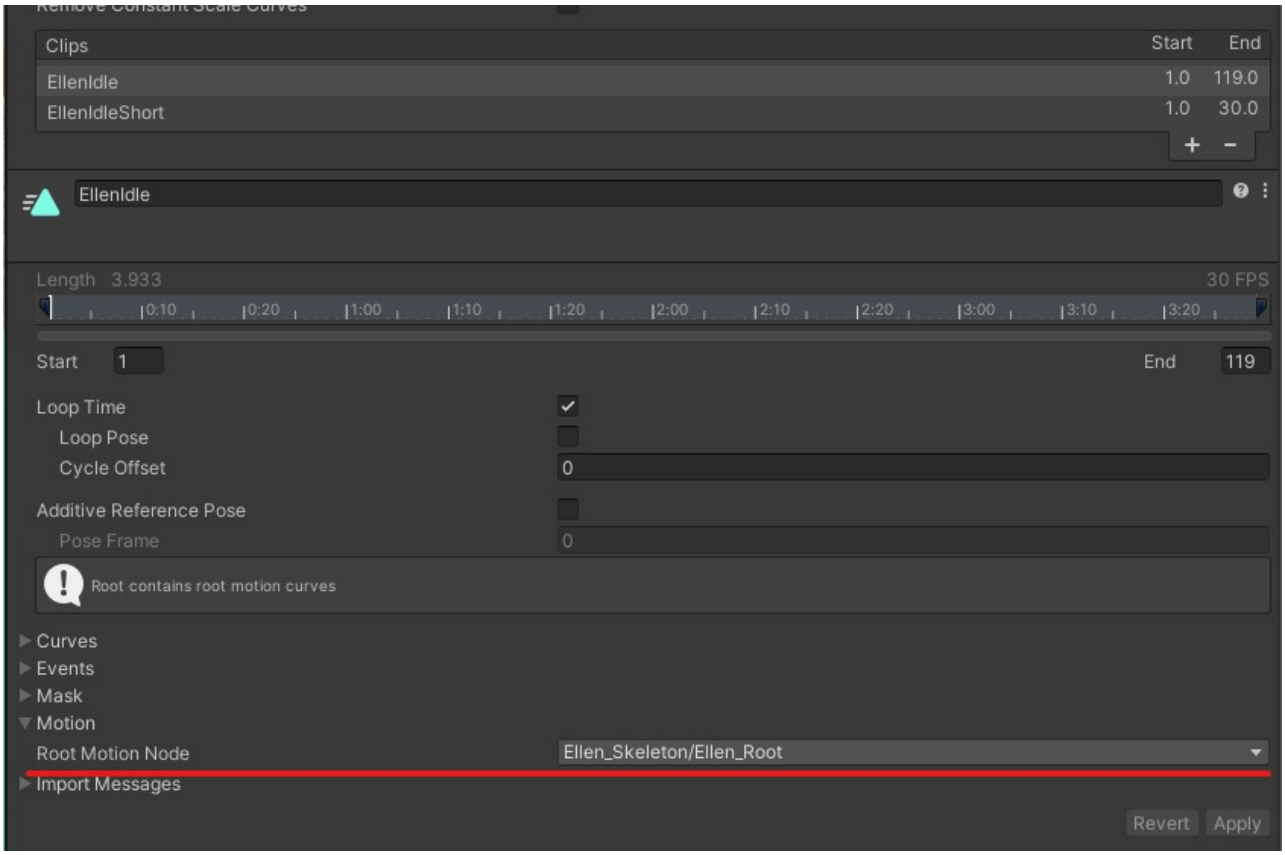
- Set the `Apply Root Motion` checkbox in the Unity `Animator` component:



- Set the `Root Bone` field in the `Rig Definition Authoring` component exactly the same as in the Unity model `Rig` importer configuration:



- Animations used in the root motion process should have a generic `Root Motion Node` defined. That node will drive animated entity position and rotation:



IMPORTANT: Root curves (‘Root T’ and ‘Root Q’ animation curves) are not supported yet.

User Curves

Rukhanka has user curves support. These curves can drive animator parameters exactly as Unity `Mekanim` does. Specify the user animation curve exactly as described in documentation. Make sure its name is the same as one of your animator controller parameters. **Rukhanka** will process these curves and writes the current value to the corresponding parameter component data.

Samples

Installation

Navigate to the `Resources/Rukhanka Animation System/Samples` directory. Import the `RukhankaSamples_HDRP.unitypackage` or `RukhankaSamples_URP.unitypackage` (depending of your rendering pipeline) package into your project. All sample scenes will be located at `Assets\RukhankaAnimationSystem\Samples`.

Basic Animation

This sample is a result of the Getting Started page of this documentation. The sample scene contains several models that play one animation each.

Bone Attachment

Entities (animated and non-animated) are handled by **Rukhanka** automatically. No extra special steps are needed. Just place your object as a child of the required bone `GameObject`. Entity hierarchy will stay intact, but **Rukhanka** will move corresponding `Entities` according to animations. This sample shows this functionality.

Animator Parameters

This sample has an animated model with a simple `Animator` created for it. Parameters that control `Animator` behavior are controlled by a simple system through UI. Controlling animator parameters from code described in Animator Parameters section of this documentation

BlendTree Showcase

Rukhanka supports all types of blend trees that Unity `Mecanim` does. This sample shows `Direct`, `1D`, `2D Simple Directional`, `2D Freeform Directional`, `2D Freeform Cartesian` blend tree types. Blend tree blend values can be controlled from in-game UI.

Avatar Mask

Avatar Masks is supported for generic (non-humanoid) animations. The use of this feature is no different than in `Unity`. Specify `Avatar Mask` and use it in Unity `Animator` to mask animation for bones. **Rukhanka** converts it into internal representation during the baking phase. This sample shows this functionality.

Multiple Blend Layers

Rukhanka has multiple animation layers support. `Additive` and `Override` layers with corresponding weights are correctly handled by **Rukhanka** runtime. In this sample, `Animator` simulates two layers represented by simple state machines.

User Curves

Custom animation curves are handled the exactly same way as they do in Unity `Mecanim`. If the animation state machine has a parameter with a name equal to the animation curve name then the value of the calculated curve at a given animation time will be copied into the parameter value. In this sample, there is an animation that has a curve whose name is the same as the animation speed parameter of the `Animator` state machine. This way animation controls its own speed. User curves are described in more detail in the User Curves section of the documentation.

Root motion

Rukhanka has limited `Root Motion` support. This sample demonstrates its use case. Root Motion features are described in detail in the corresponding section of documentation.

Animator Override Controller

Unity `Animator` has a feature called Animtor Override Controller. This feature enables to use of a different set of animations for a given preconfigured `Animator`. This feature is also supported by **Rukhanka**. This sample has an `Animator Controller` and corresponding `Animator Override Controller` which overrides several animations.

Non-Skinned Mesh Animation

Rukhanka can animate arbitrary `Entity` hierarchy with user-defined animation. This sample shows this use case. Refer to the Non-skinned Meshes page for a detailed description of this feature.

Crowd

This sample shows **Rukhanka** ability to animate a big number of different animated models. A simple prefab spawner system is used to spawn big counts of prebaked animated prefabs.

Stress Test

This sample is basically the same as the `Crowd` sample but with all skinned mesh models replaced by plain cubes. This step removes the big graphics pipeline pressure of the `Crowd` scene and keeps only raw **Rukhanka** animation system performance. This sample scene can be used for checking animation performance limits for tested systems/hardware.

Tips

Perfomance Optimization Tips

There are several performance optimization tips for **Rukhanka**:

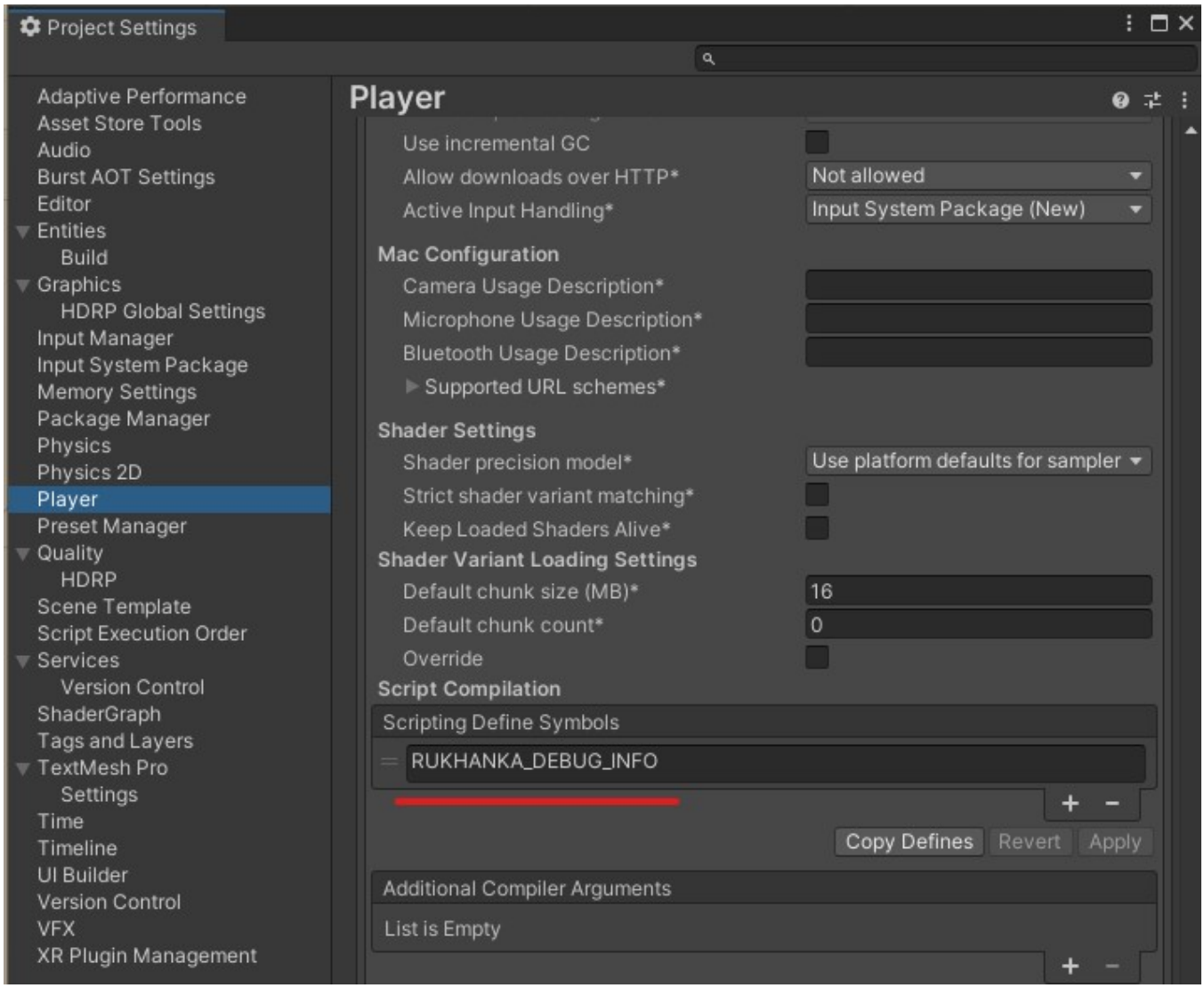
- Lower total bone count. **Rukhanka** splits animation workload per bone and not per entity. This allows full utilize all available processor power for low bone count meshes as well as high bone count. So overall performance will depend on total bone count linearly.
- Additive animations are as twice slow as normal ones. Heavy use of additive animations will be slower than ordinary.
- Do not forget to disable `RUHANKA_DEBUG_INFO` during performance tests.

Debug and Validation

Extended Validation Layer

Despite that the animation system heavily depends on name relations between components (bone names, animation parameter names, state machine state names, etc), `string` values are used only in bake time. Bake systems convert all `string` values into `Hash128` representations and work with them in runtime. No string data is available during state machines and animation processing. This approach is very performant but debugging and validation in case of issues become very complicated.

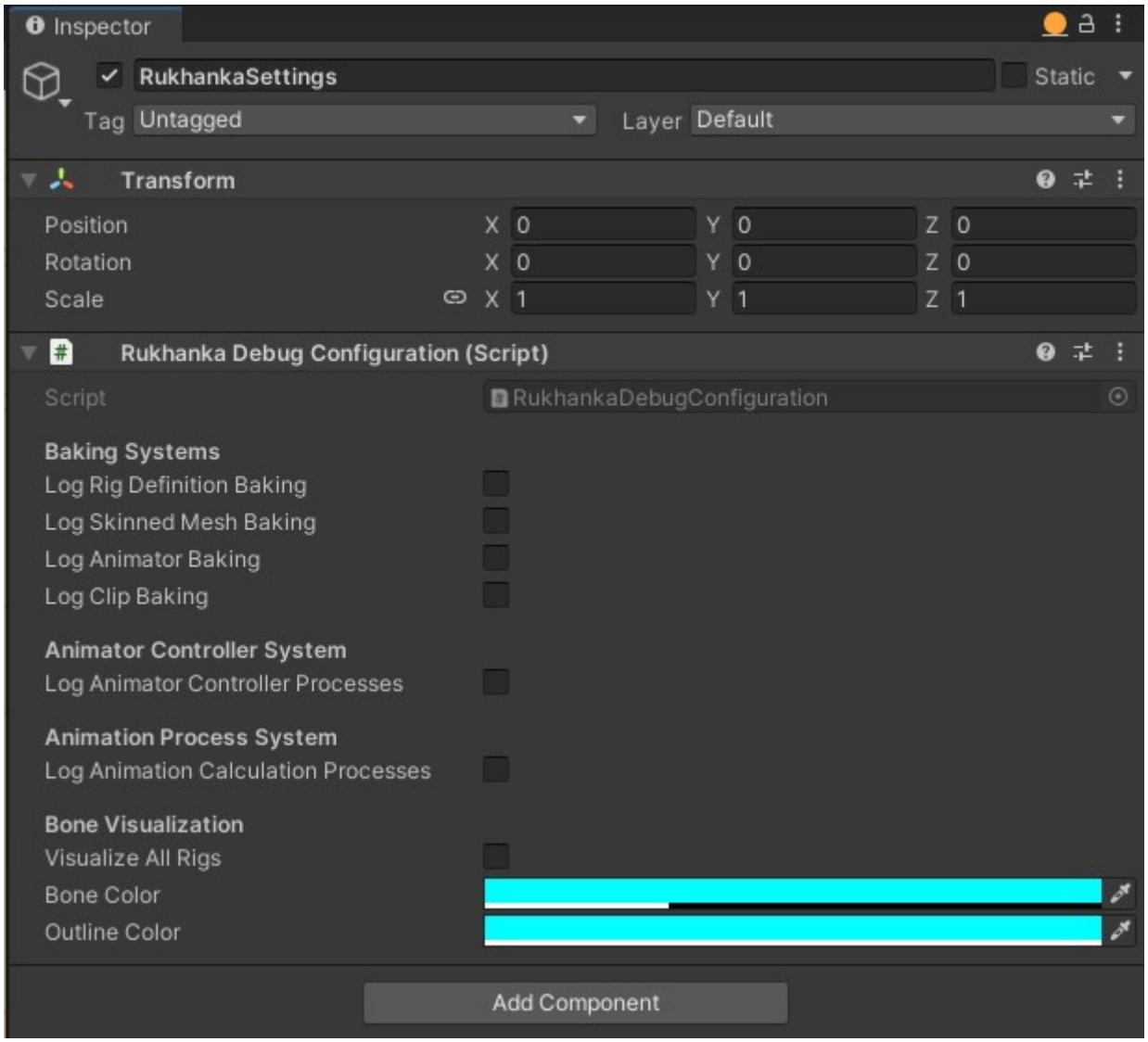
To make the easier process of watching for state and parameter changing, debugging, and detailed logging of baking processes, **Rukhanka** introduces a special `extended validation` mode. This mode can be enabled by adding the `RUHANKA_DEBUG_INFO` script definition symbol into project preferences:



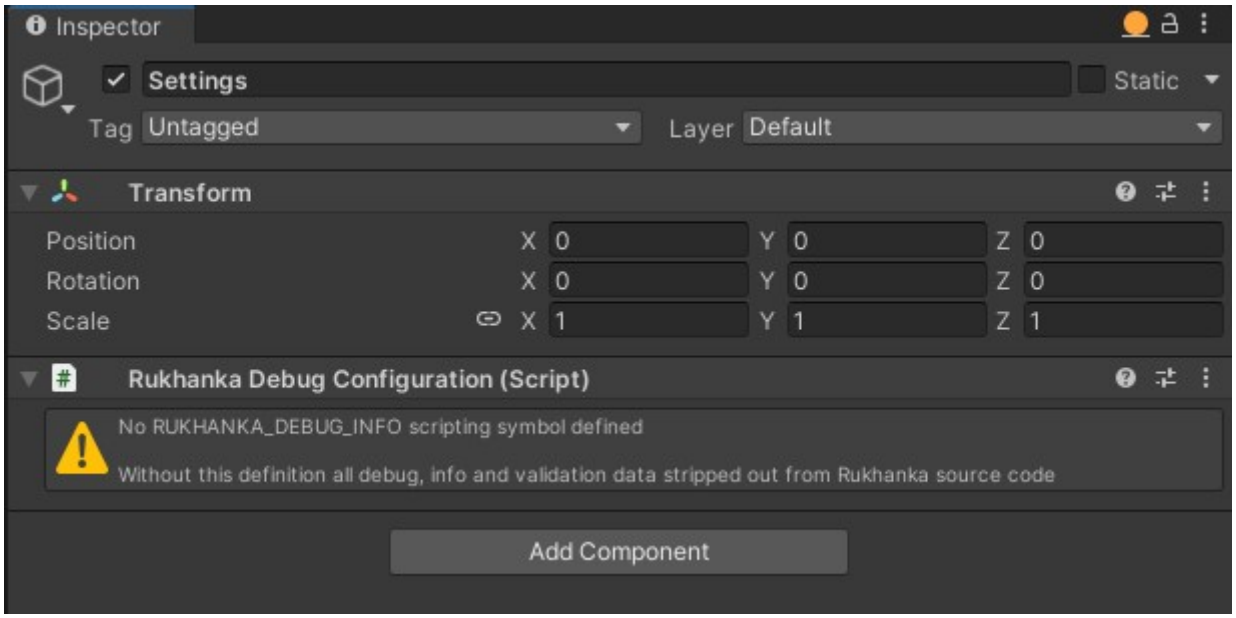
Adding this symbol, **Rukhanka** will add to all internal structures its corresponding string fields (FixedString or BlobString for `Burst` compatibility where appropriate). Watching these members in the debugger and logging makes it much easier to investigate and fix problems in animations

Logging capabilities

By defining `RUHANKA_DEBUG_INFO` extended logging and visualization capabilities have also become available. To configure them add the `Rukhanka Debug Configuration` authoring component to any `GameObject` inside `Entities Subscene`.



If `RUHANKA_DEBUG_INFO` is not defined this configuration script will show a warning message and no configuration options will be available:



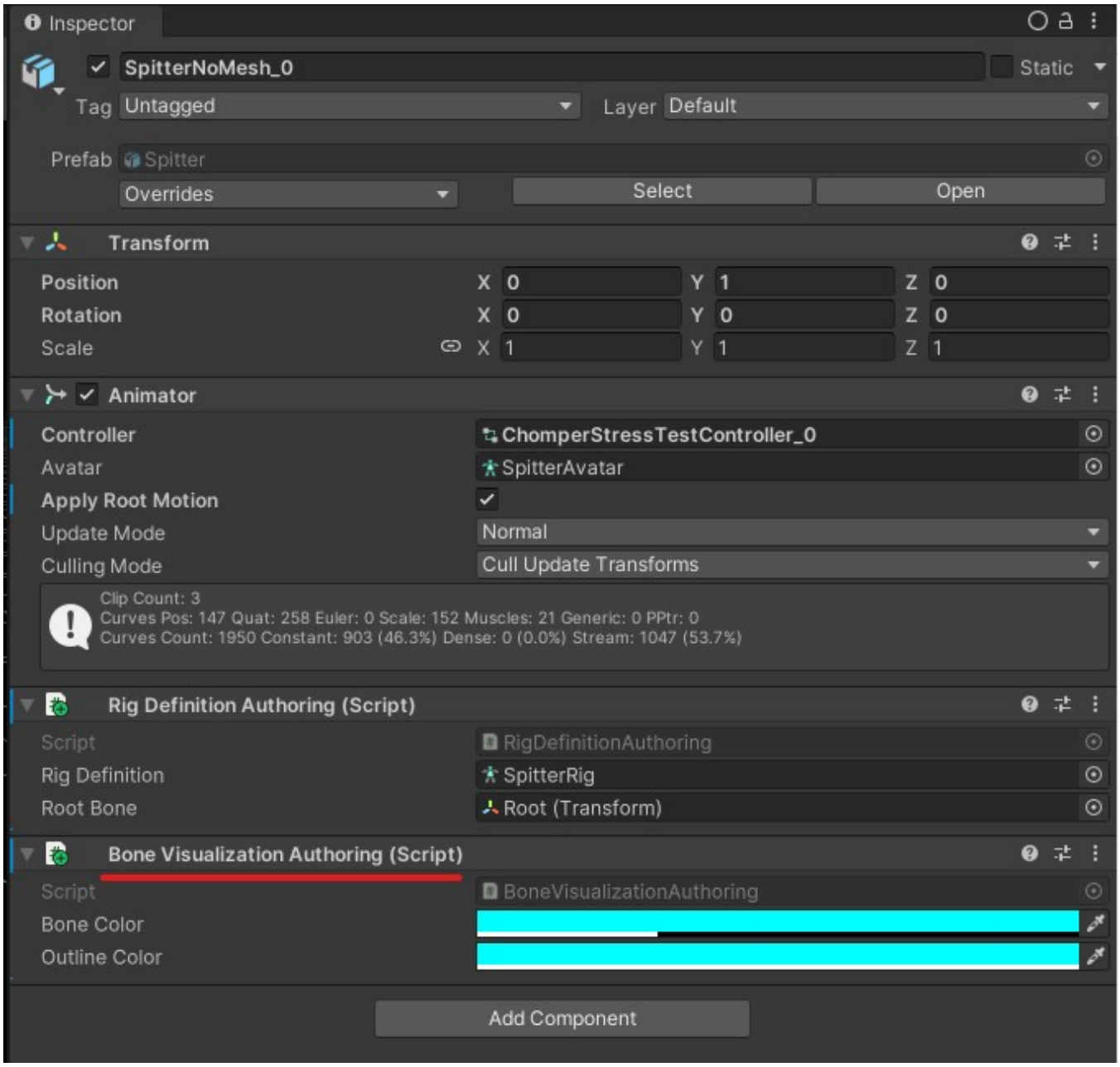
- *Baking Systems* logging will log a total of `Authoring Components` baked as well as additional warnings and messages during the baking process.

- *Animator Controller System* logging will enable the log of animator controller internal state changes and additional details.
- _Animation Process System__ logging will enable the log of animation core internal details during runtime.
- *Bone Visualization* enables internal bone renderer for all **Rukhanka** `Rigs`.

Bone Visualization

There are two options to enable *Bone Visualization* capability for **Rukhanka** `Rig`:

1. Enable bone visualization for all meshes in the scene. This is done by the checkbox described in the previous section on this page.
2. Add the `Bone Visualization Authoring` component to the required animated object. Note that this way bone visualization will work even without `RUHANKA_DEBUG_INFO` defined.

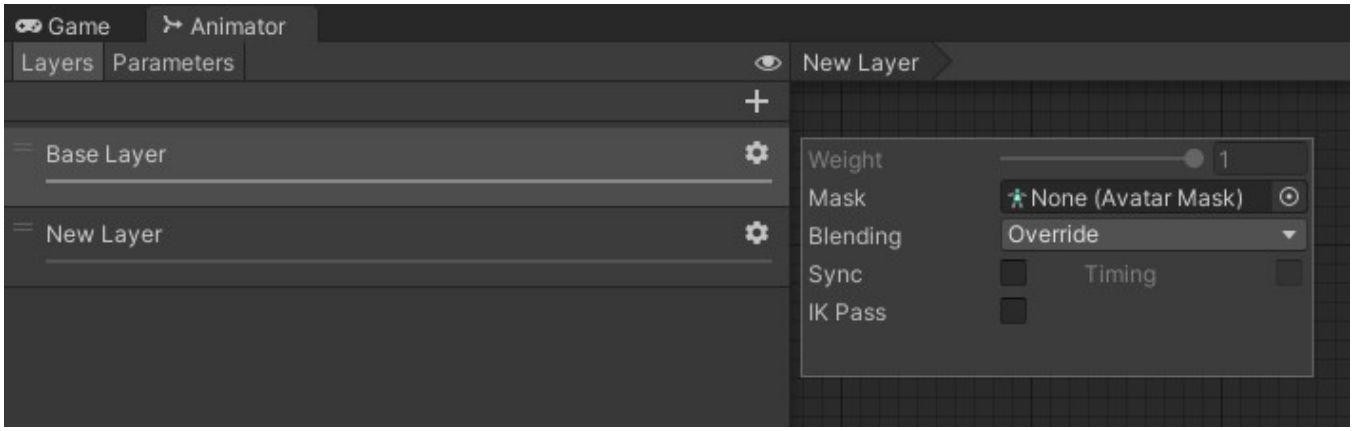

Inspector
SpitterNoMesh_0
Static
Tag Untagged
Layer Default
Prefab Spitter
Overrides
Select
Open
Transform
Position X 0 Y 1 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Animator
Controller ChomperStressTestController_0
Avatar SpitterAvatar
Apply Root Motion
Update Mode Normal
Culling Mode Cull Update Transforms
Clip Count: 3
Curves Pos: 147 Quat: 258 Euler: 0 Scale: 152 Muscles: 21 Generic: 0 PPtr: 0
Curves Count: 1950 Constant: 903 (46.3%) Dense: 0 (0.0%) Stream: 1047 (53.7%)
Rig Definition Authoring (Script)
Script RigDefinitionAuthoring
Rig Definition SpitterRig
Root Bone Root (Transform)
Bone Visualization Authoring (Script)
Script BoneVisualizationAuthoring
Bone Color
Outline Color
Add Component

Chengelog

[1.0.0] - Initial release

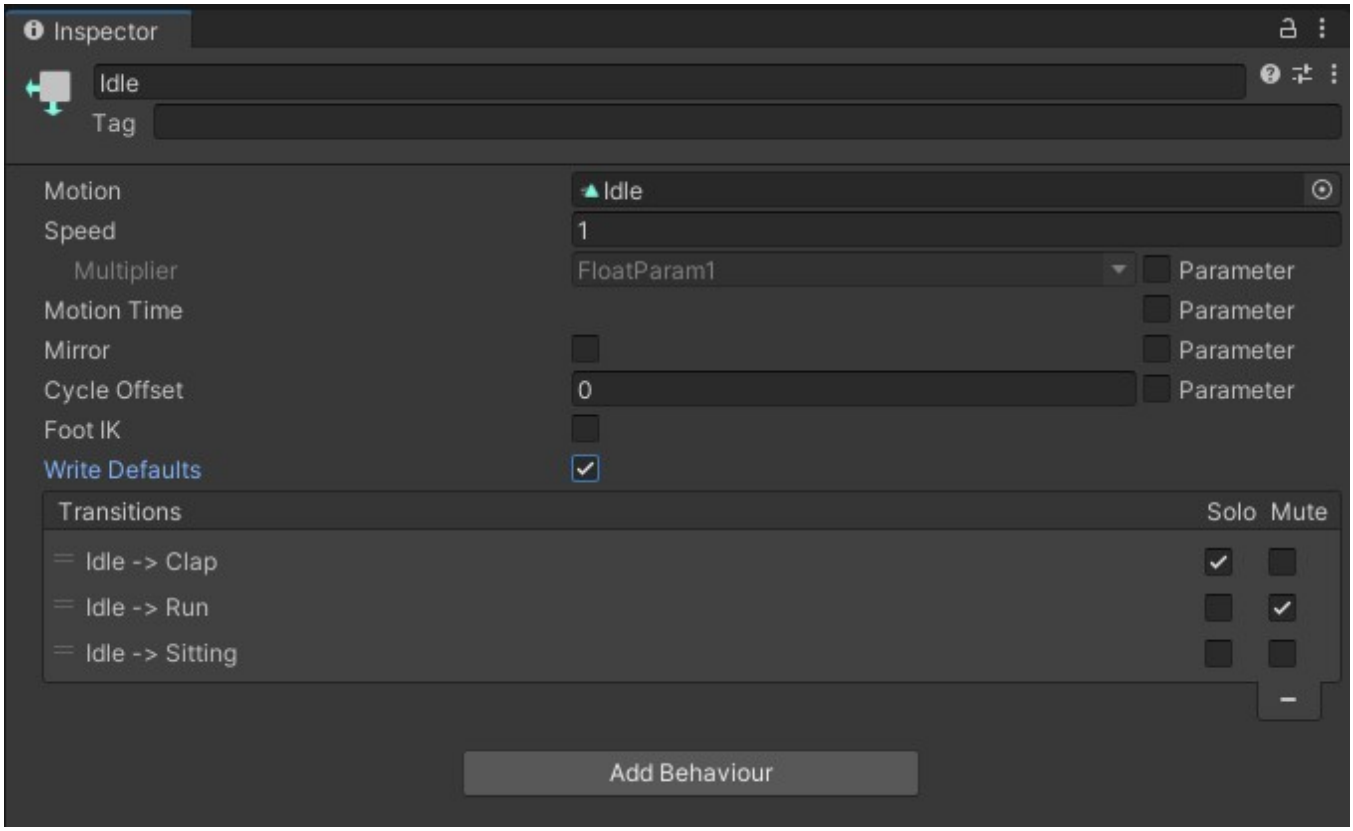
Feature Support Tables

Animator Controller Layer



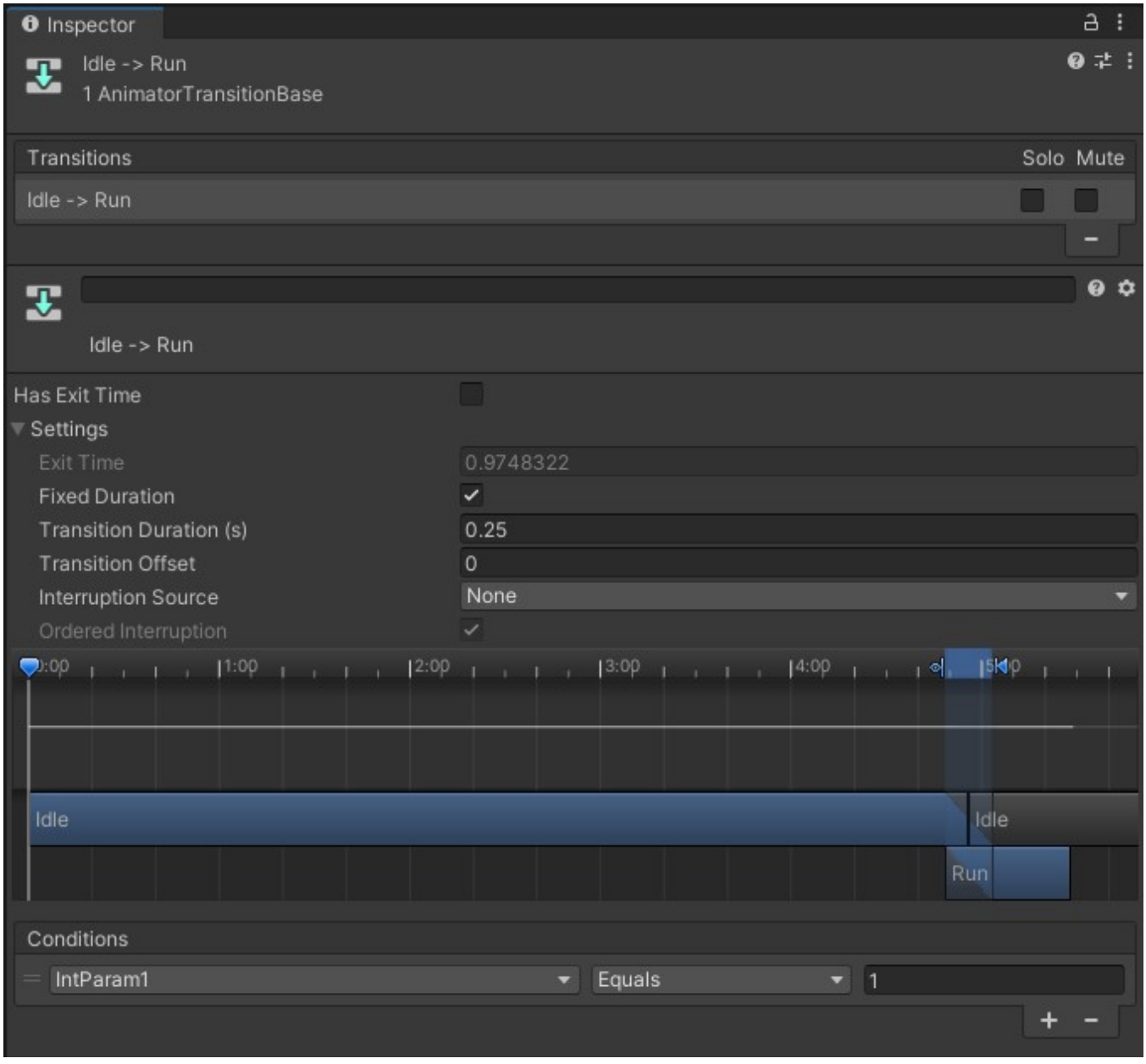
Feature Name	Support Status	Additional Notes
Multiple Layers	+	
Sub-State Machines	+	
Weight	+	
Mask	+	
Override Blending	+	
Additive Blending	+	
Sync	-	
IK Pass	-	

Animator State



Feature Name	Support Status	Additional Notes
Motion	+	
Speed	+	
Speed Multiplier	+	
Motion Time	+	
Mirror	-	
Cycle Offset	+	
Foot IK	-	
Write Defaults	-	

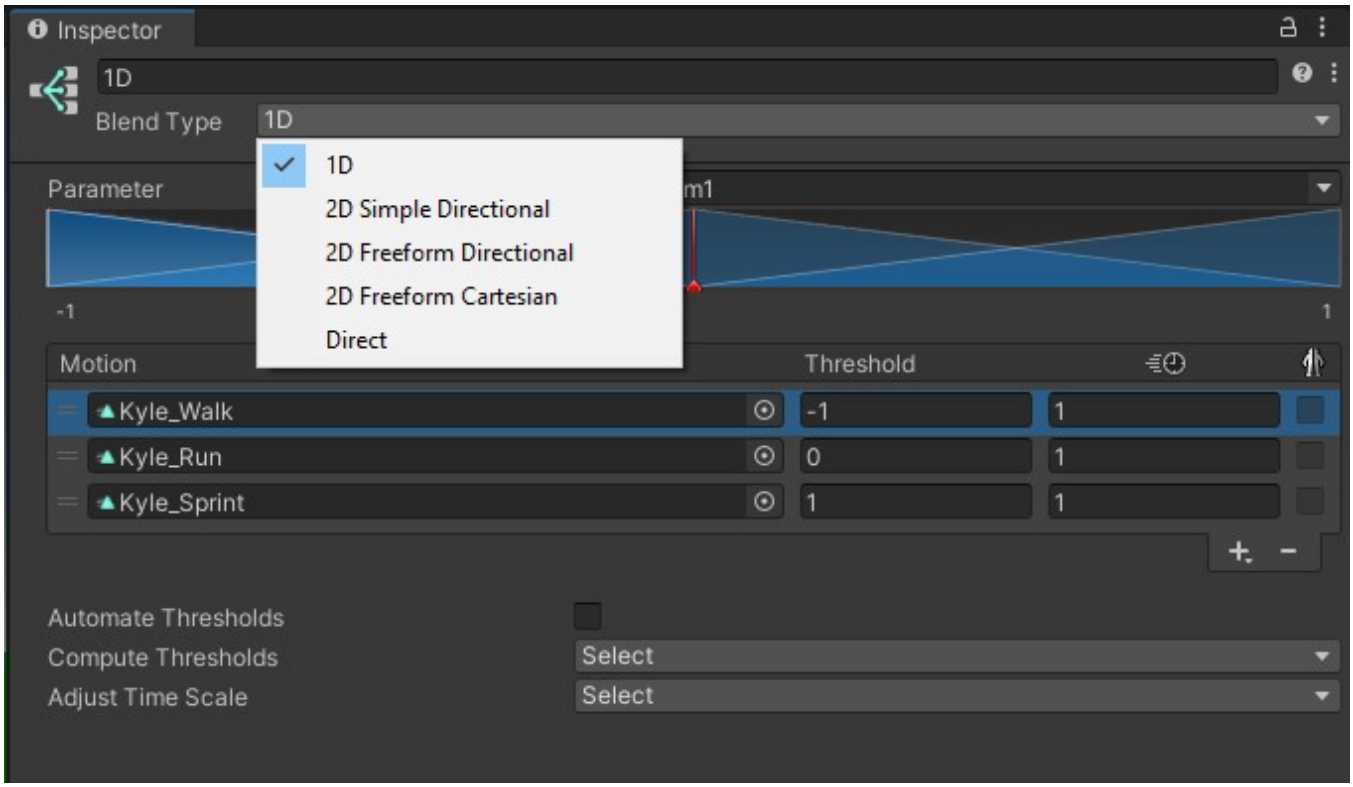
Animator Transition



Feature Name	Support Status	Additional Notes
Solo	+	
Mute	+	
Has Exit Time	+	
Exit Time	+	
Fixed Duration	+	

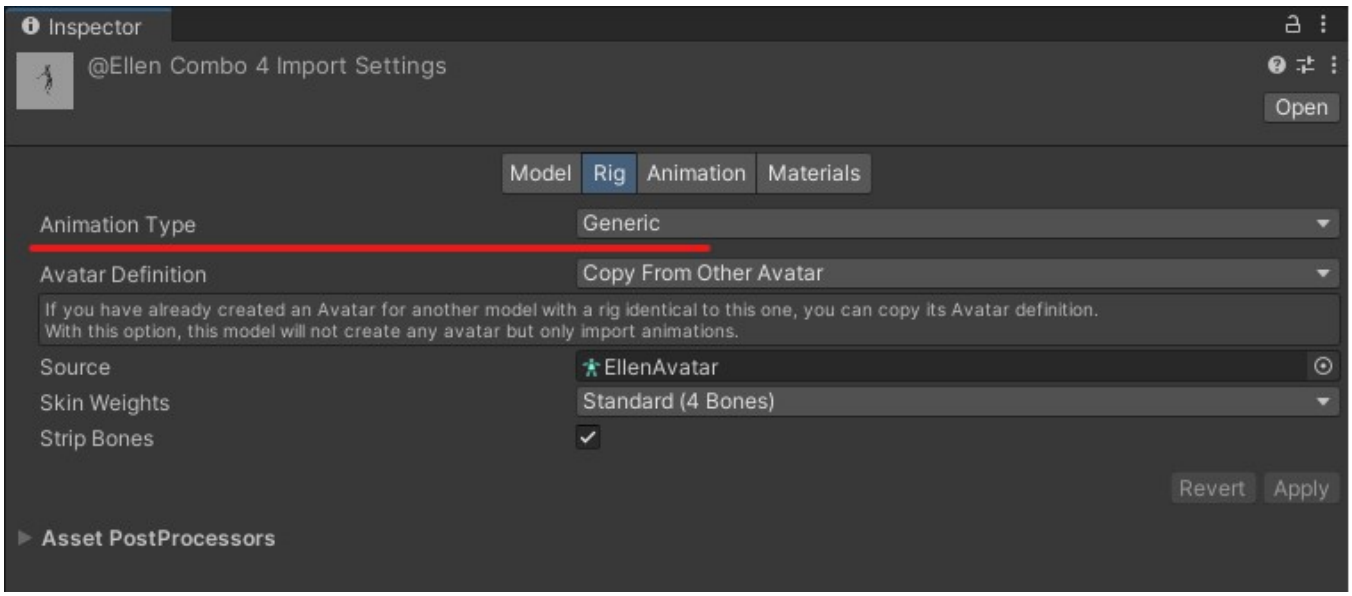
Feature Name	Support Status	Additional Notes
Transition Duration	+	
Transition Offset	+	
Interruption Source	-	
Ordered Interruption	-	
Can Transition To Self	+	Available only in `Any State`
Int Conditions	+	
Float Conditions	+	
Bool Conditions	+	
Trigger Conditions	+	

Blend Tree Features



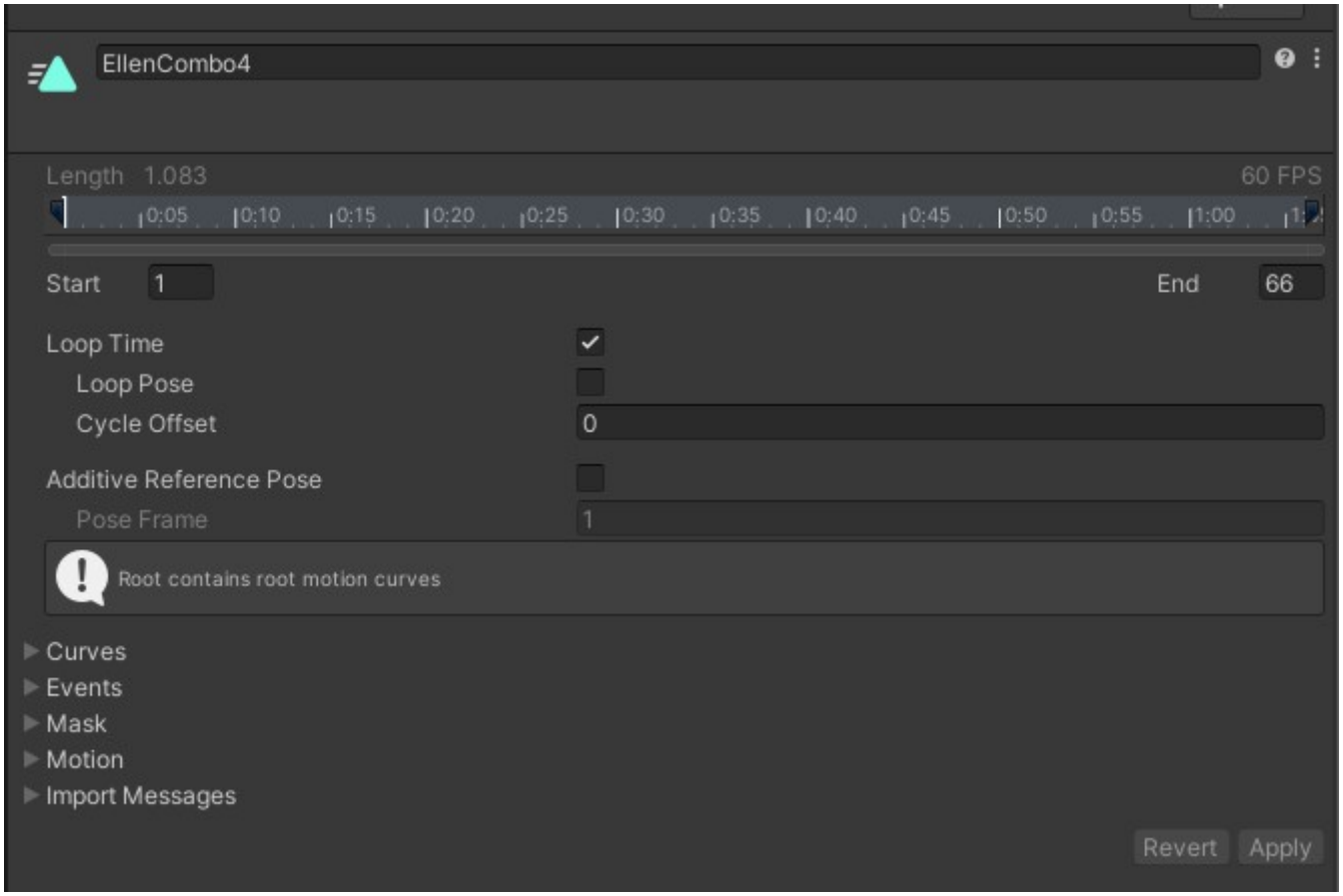
Feature Name	Support Status	Additional Notes
1D	+	
2D Simple Directional	+	
2D Freeform Directional	+	
2D Freeform Cartesian	+	
Direct	+	
Automate Thresholds	o	Not handled by **Rukhanka**
Compute Thresholds	o	Not handled by **Rukhanka**
Adjust Time Scale	o	Not handled by **Rukhanka**

Animation Rig Properties



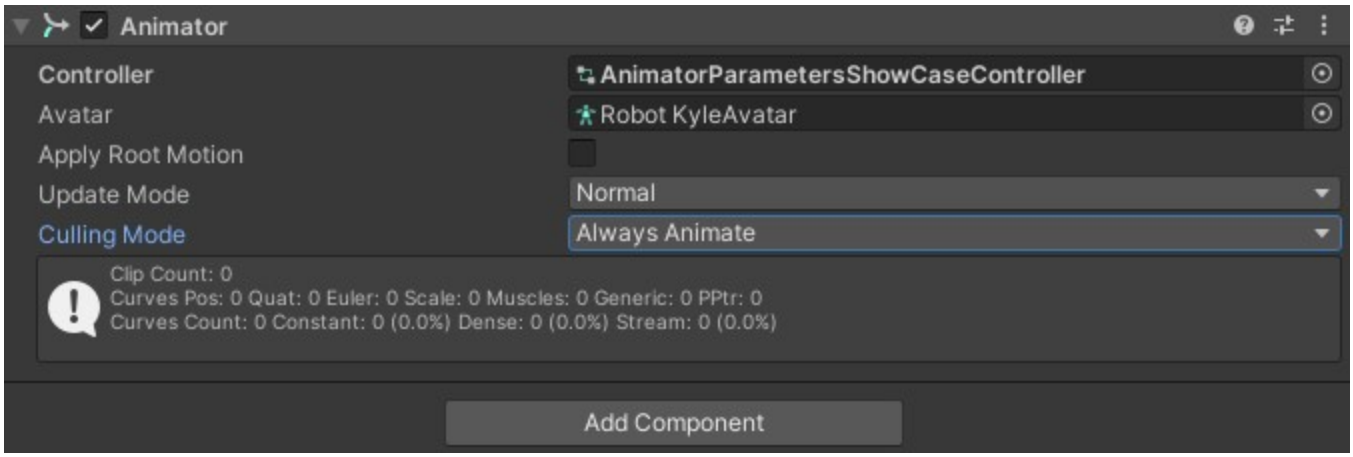
Feature Name	Support Status	Additional Notes
Animation Type *Generic*	+	
Animation Type *Legacy*	-	
Animation Type *Humanoid*	-	

Animation Properties



Feature Name	Support Status	Additional Notes
Loop Time	+	
Loop Pose	+	
Cycle Offset	+	
Additive Reference Pose	+	
Pose Frame	+	
Curves	+	Documentation
Events	-	
Mask	-	
Motion	+	Documentation
Generic Root Curves	-	`Root T` and `Root Q`

Animator Features



Feature Name	Support Status	Additional Notes
Controller	+	
Avatar	-	Need to be configured separately
Apply Root Motion	o	Partial. Read the docs
Update Mode	-	
Culling Mode	-	