Rukhanka ECS Animation System

Table of contents:

RUKHANKA

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.

'Netcode for Entities' package Support

Rukhanka supports animation synchronization between server and clients in network games by working with 'Netcode for Entities' ECS library. Animation synchronization can be done by using predicted and interpolated ghosts. Client-only entities can coexist together with network-synchronized ones.

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

Upgrading Rukhanka

[1.2.x -> 1.3.0]

- The rig definition avatar mask asset is not needed anymore. Prepare your model to contain all necessary information by following the rig definition setup process. All previously created rig definition avatar mask assets can be safely deleted.

Getting Started

Prerequisites

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

- Unity 2022.3.0f1+
- Unity `Entities` package version 1.0.16+ (installed automatically as dependency)
- Unity `Entities.Graphics` package version 1.0.16+ (installed automatically as dependency)
- HDRP or URP as required by `Entities.Graphics` package
- [Optional] `Netcode for Entities` package for network animation synchronization support

Animated Object Setup

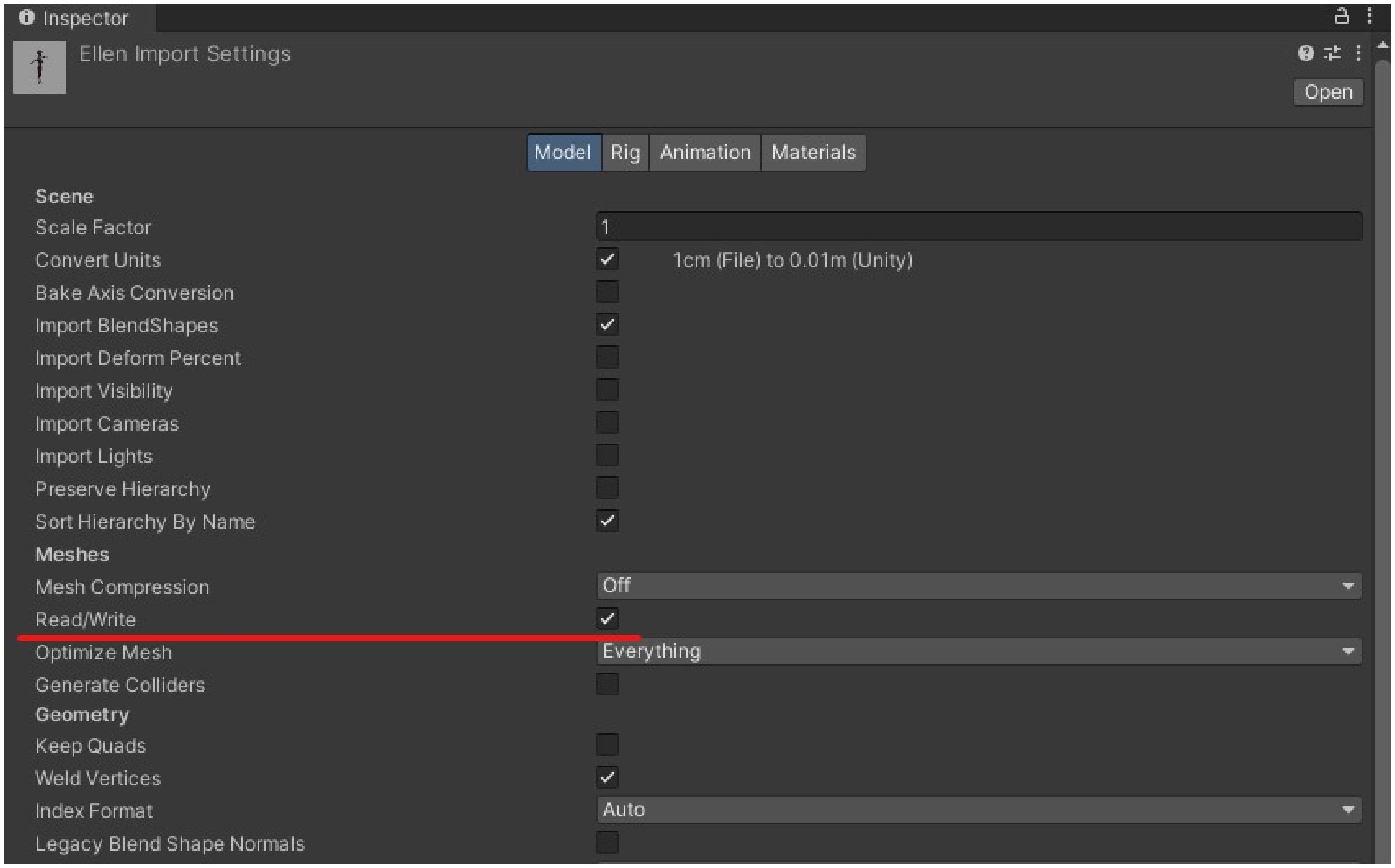
To make animations work correctly there are some preparation setup steps are required.

Model Importer

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

1. Model Importer Tab.

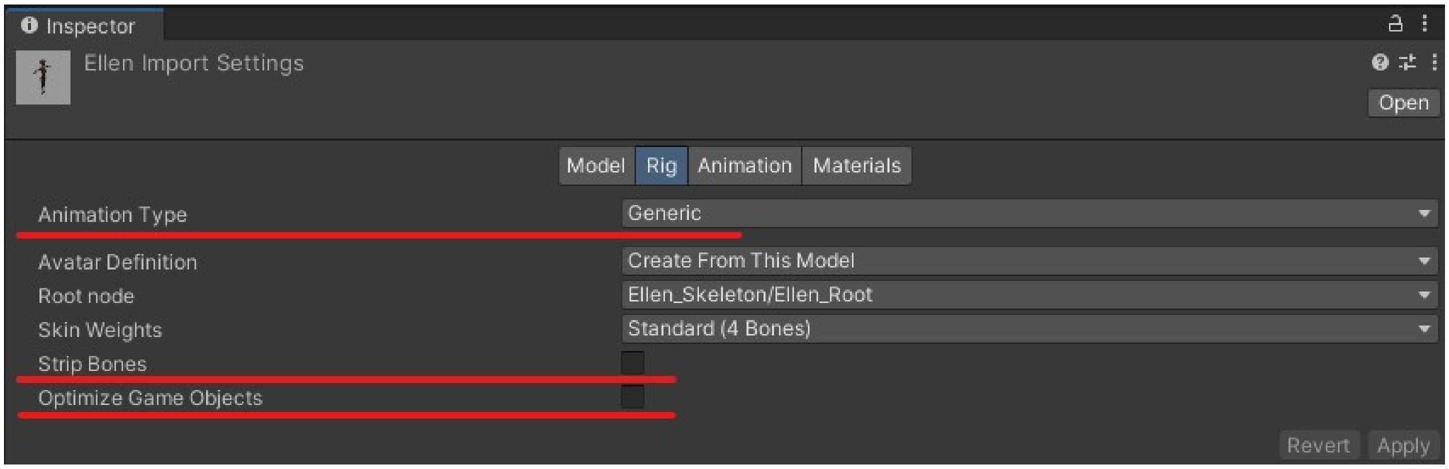
    - Enable `Read/Write` property for the mesh. This is requirement of `Deformation Subsystem` of `Entities.Graphics` package.


Inspector
Ellen Import Settings
Open
Model | Rig | Animation | Materials

Scene	
Scale Factor	1
Convert Units	✓ 1cm (File) to 0.01m (Unity)
Bake Axis Conversion	
Import BlendShapes	✓
Import Deform Percent	
Import Visibility	
Import Cameras	
Import Lights	
Preserve Hierarchy	
Sort Hierarchy By Name	✓
Meshes	
Mesh Compression	Off
Read/Write	✓
Optimize Mesh	Everything
Generate Colliders	
Geometry	
Keep Quads	
Weld Vertices	✓
Index Format	Auto
Legacy Blend Shape Normals	

2. Rig Importer Tab.

- Set `Animation Type` to `Generic` or `Humanoid`.
- 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.


Inspector
Ellen Import Settings
Open
Model | Rig | Animation | Materials

Animation Type	Generic
Avatar Definition	Create From This Model
Root node	Ellen_Skeleton/Ellen_Root
Skin Weights	Standard (4 Bones)
Strip Bones	
Optimize Game Objects	

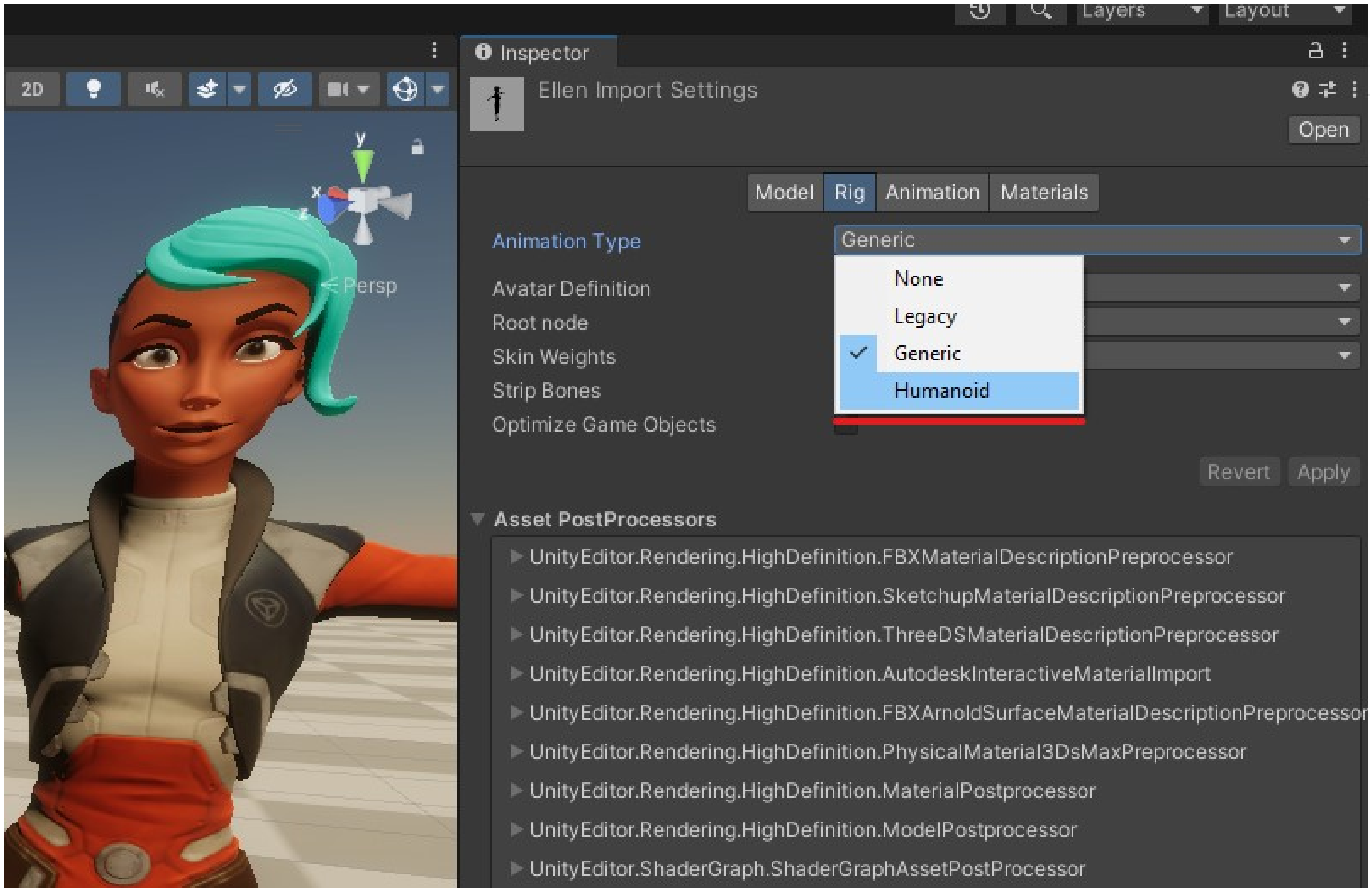
Revert Apply

Rig Definition

Rukhanka can get all required information about skeleton rig structure from Unity `Avatar`. There are several required simple steps to populate Unity `Avatar` with all necessary rig data:

1. In the importer window of your animated model switch `Animation Type` to the `Humanoid` and press `Apply`.

   > ⚠ **IMPORTANT**
   >
   > You must do this for every model (even `Generic`). By switching to the `Humanoid` animation type Unity generates publicly available information about the skeleton rig structure that **Rukhanka** reads.
   >
   > You will get an error during the baking process if avatar rig information is not available to **Rukhanka**.



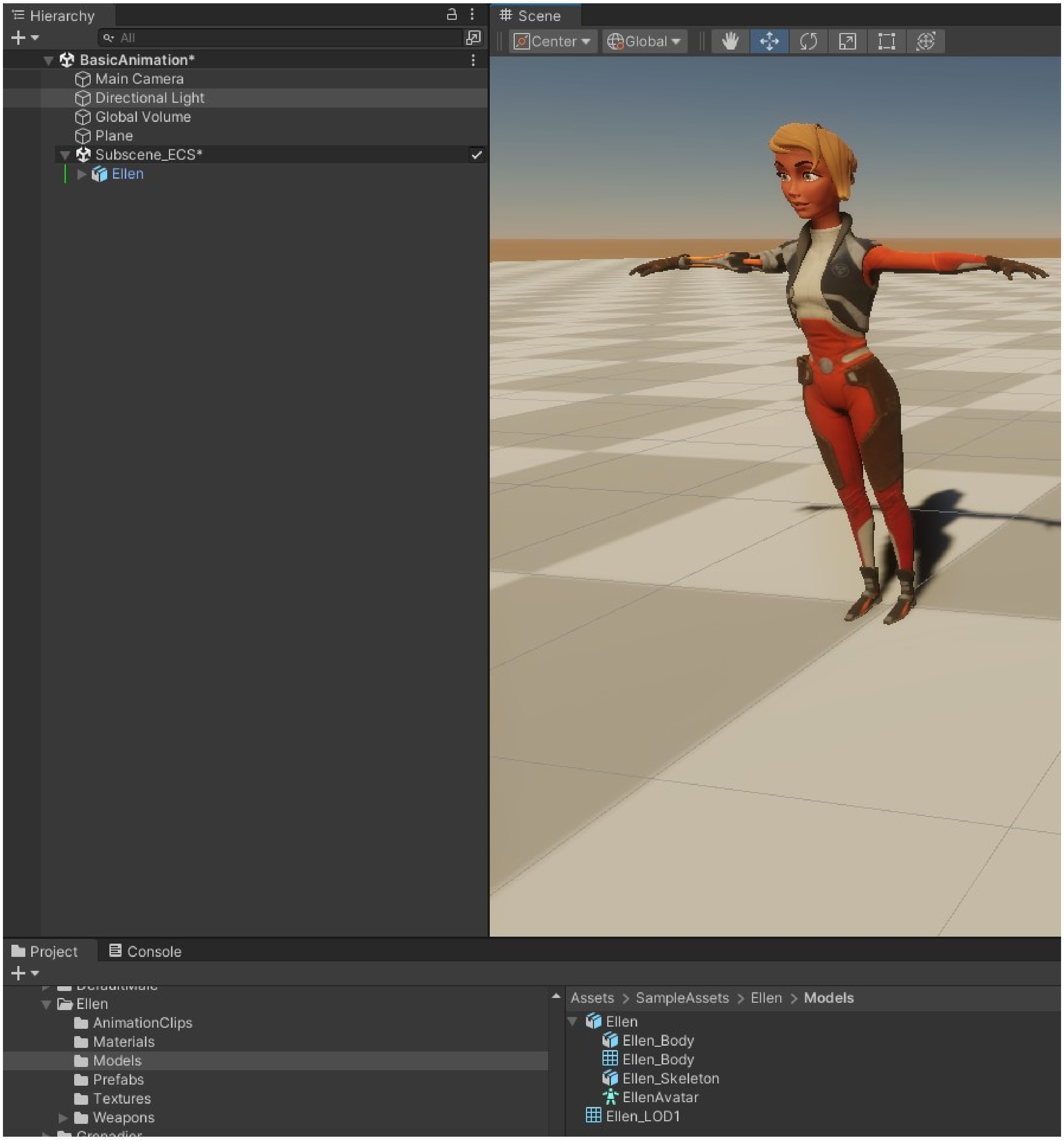
UnityEditor.Rendering.HighDefinition.FBXMaterialDescriptionPreprocessor
UnityEditor.Rendering.HighDefinition.SketchupMaterialDescriptionPreprocessor
UnityEditor.Rendering.HighDefinition.ThreeDSMaterialDescriptionPreprocessor
UnityEditor.Rendering.HighDefinition.AutodeskInteractiveMaterialImport
UnityEditor.Rendering.HighDefinition.FBXArnoldSurfaceMaterialDescriptionPreprocessor
UnityEditor.Rendering.HighDefinition.PhysicalMaterial3DsMaxPreprocessor
UnityEditor.Rendering.HighDefinition.MaterialPostprocessor
UnityEditor.Rendering.HighDefinition.ModelPostprocessor
UnityEditor.ShaderGraph.ShaderGraphAssetPostProcessor

2. For `Humanoid` models configuration process is ended already. For `Generic` models you need to switch `Animation Type` of model back to `Generic` and press `Apply`.

Authoring Object Setup

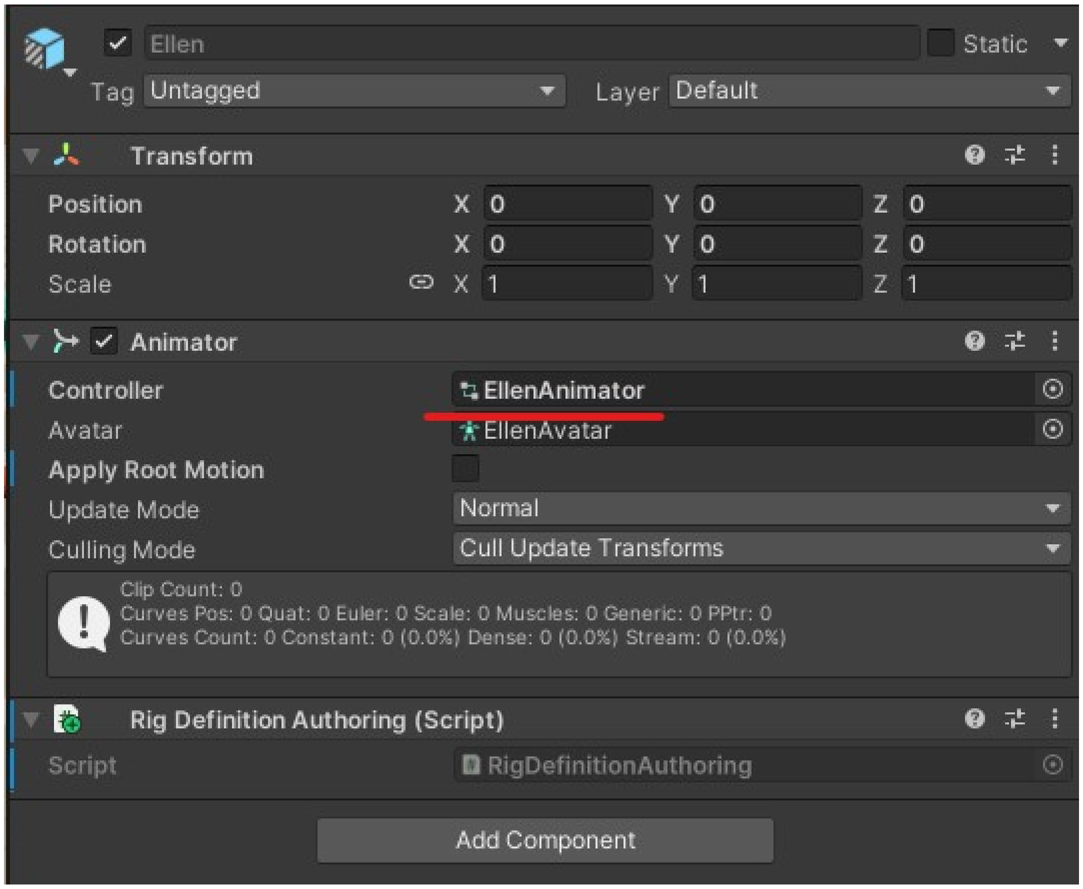
Final step is to create authoring `GameObject` inside `Entities` Subscene

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



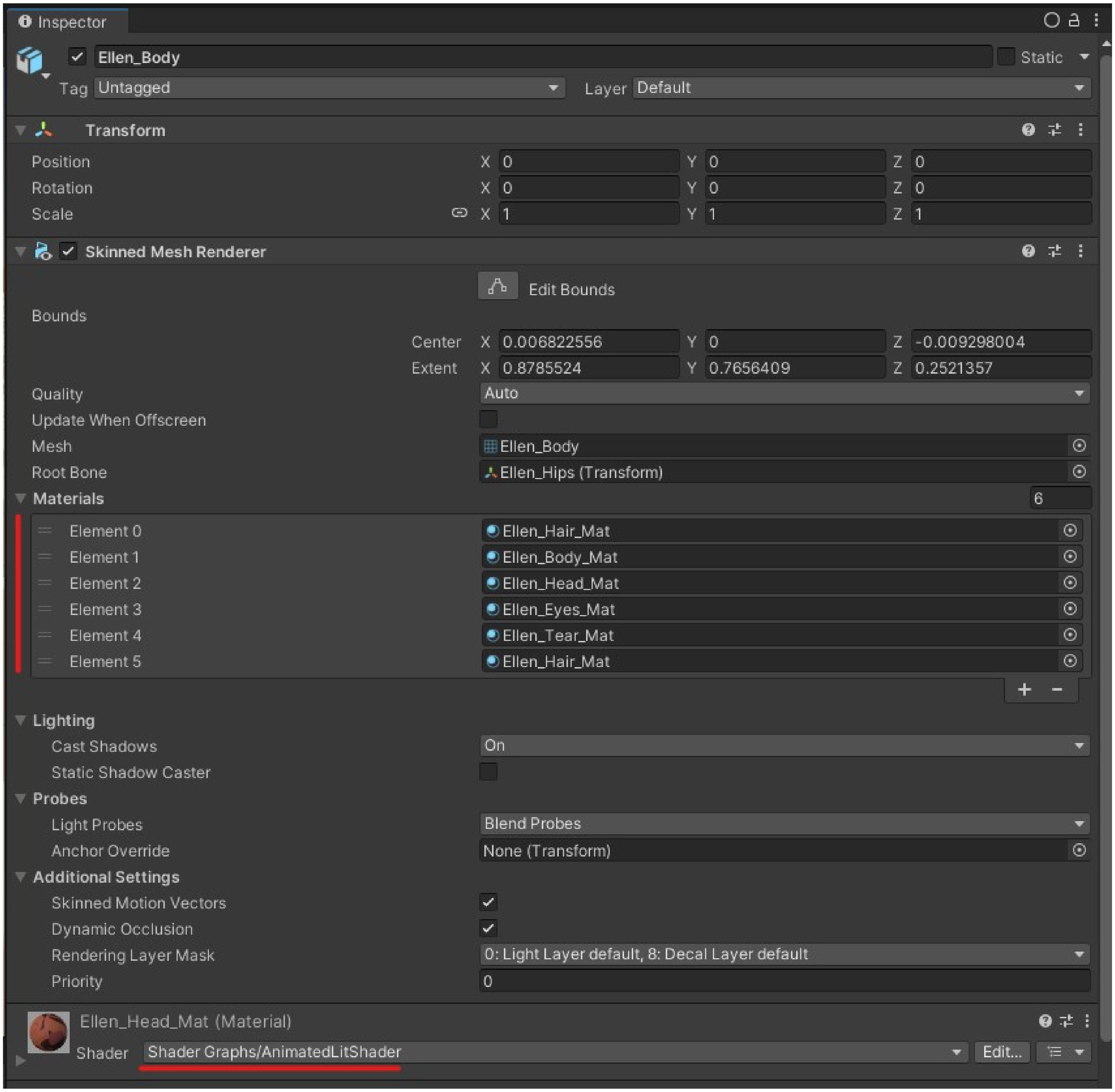
2. Add `Rig Definition Authoring` component to newly created object.

3. Create standard `Animator Controller` and fill it as you wish (one state with one animation will be a good start).


Ellen
Static
Tag Untagged
Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Animator
Controller EllenAnimator
Avatar EllenAvatar
Apply Root Motion
Update Mode Normal
Culling Mode Cull Update Transforms
Clip Count: 0
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0 Generic: 0 PPtr: 0
Curves Count: 0 Constant: 0 (0.0%) Dense: 0 (0.0%) Stream: 0 (0.0%)
Rig Definition Authoring (Script)
Script RigDefinitionAuthoring

Shaders And Materials

Rukhanka does not render 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 using `Unity Shader Graph` or `Amplify Shader Editor` as described in the Shaders with Deformation page of this manual. Make and assign all required materials to your animated model.


Inspector

✓ Ellen_Body — Static

Tag: Untagged — Layer: Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

✓ Skinned Mesh Renderer

Edit Bounds

Bounds

	X	Y	Z
Center	0.006822556	0	-0.009298004
Extent	0.8785524	0.7656409	0.2521357

Quality	Auto
Update When Offscreen	
Mesh	Ellen_Body
Root Bone	Ellen_Hips (Transform)
Materials	6
Element 0	Ellen_Hair_Mat
Element 1	Ellen_Body_Mat
Element 2	Ellen_Head_Mat
Element 3	Ellen_Eyes_Mat
Element 4	Ellen_Tear_Mat
Element 5	Ellen_Hair_Mat

Lighting

Cast Shadows	On
Static Shadow Caster	

Probes

Light Probes	Blend Probes
Anchor Override	None (Transform)

Additional Settings

Skinned Motion Vectors	✓
Dynamic Occlusion	✓
Rendering Layer Mask	0: Light Layer default, 8: Decal Layer default
Priority	0

Ellen_Head_Mat (Material)

Shader: Shader Graphs/AnimatedLitShader — Edit...

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

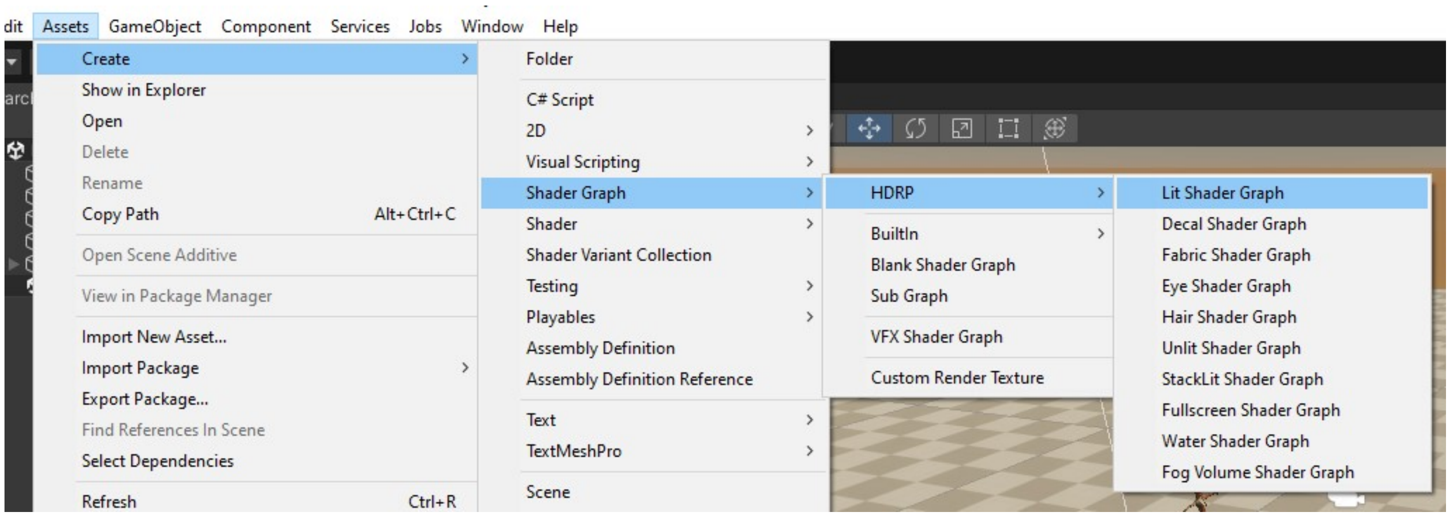
Shaders with Deformations

For the correct rendering of skinned meshes deformed by **Rukhanka**, an ECS deformation-aware shader should be created. To make this task `Unity Shader Graph` or `Amplify Shader Editor` tool can be used.

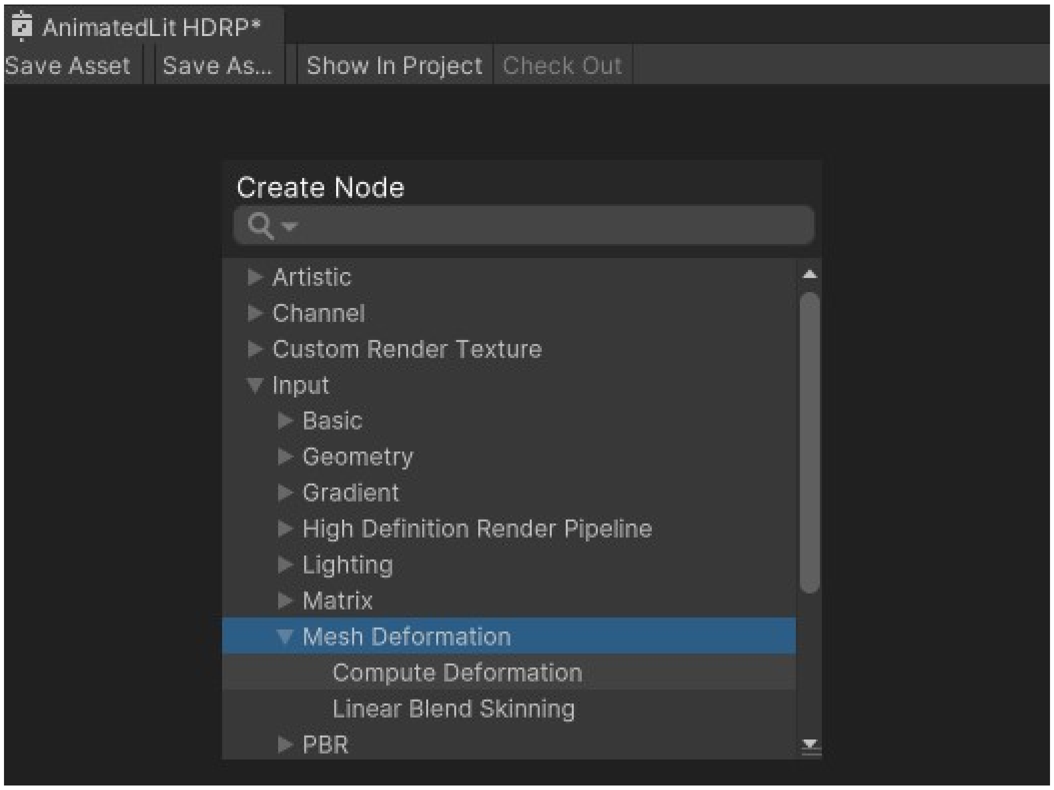
Unity Shader Graph

Creating deformation-compatible shaders using `Unity Shader Graph` is straightforward. The whole process is described in detail in mesh deformation section of official `Entities Graphics` documentation. The process consists of several simple steps:

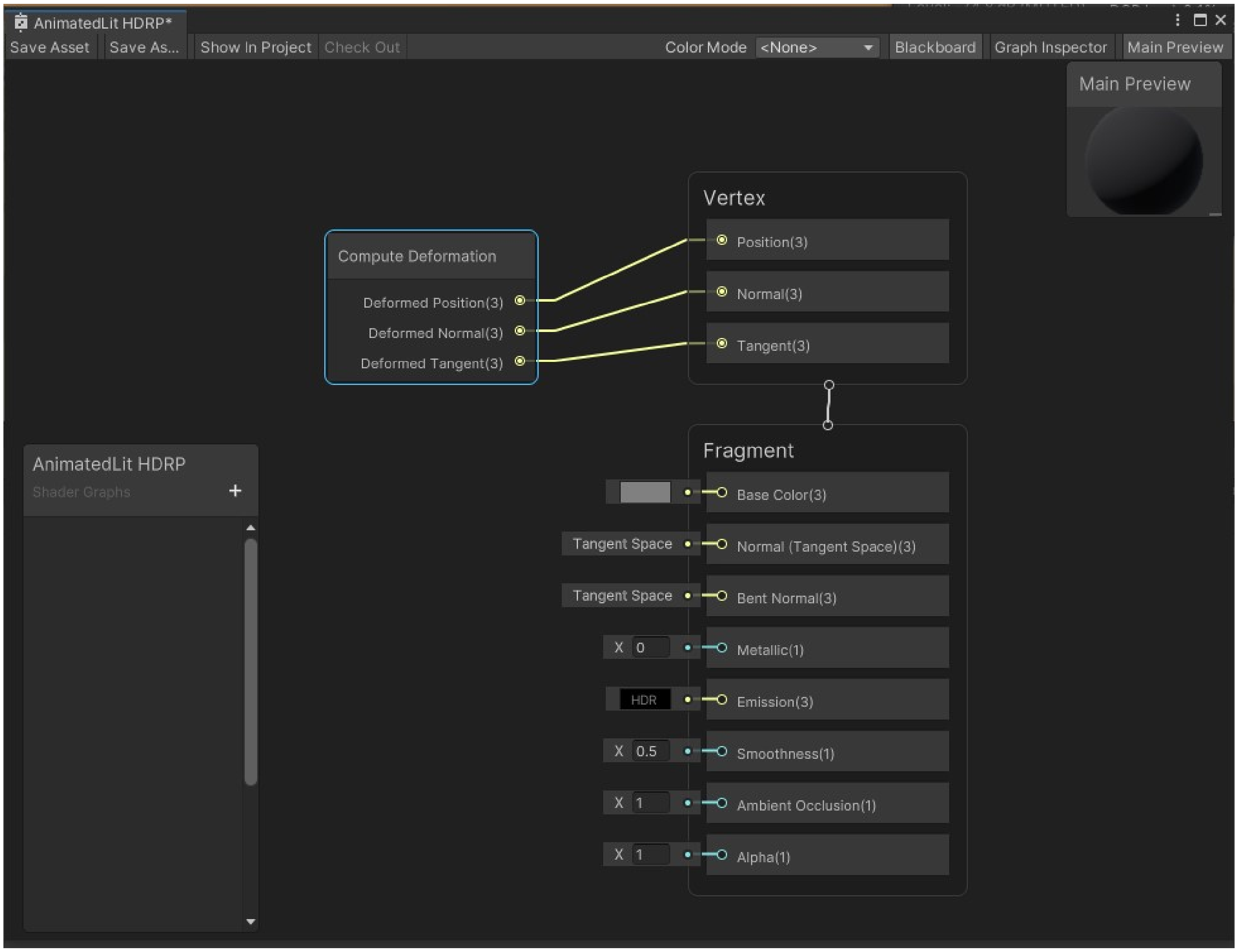
1. Create a shader graph (URP or HDRP depending on the render pipeline you are using) and open it for editing.



2. Add the `Compute Deformation` node to the Shader Graph.



3. Connect position, normal, and tangent output ports for the `Compute Deformation` node to the corresponding input ports of the master node.

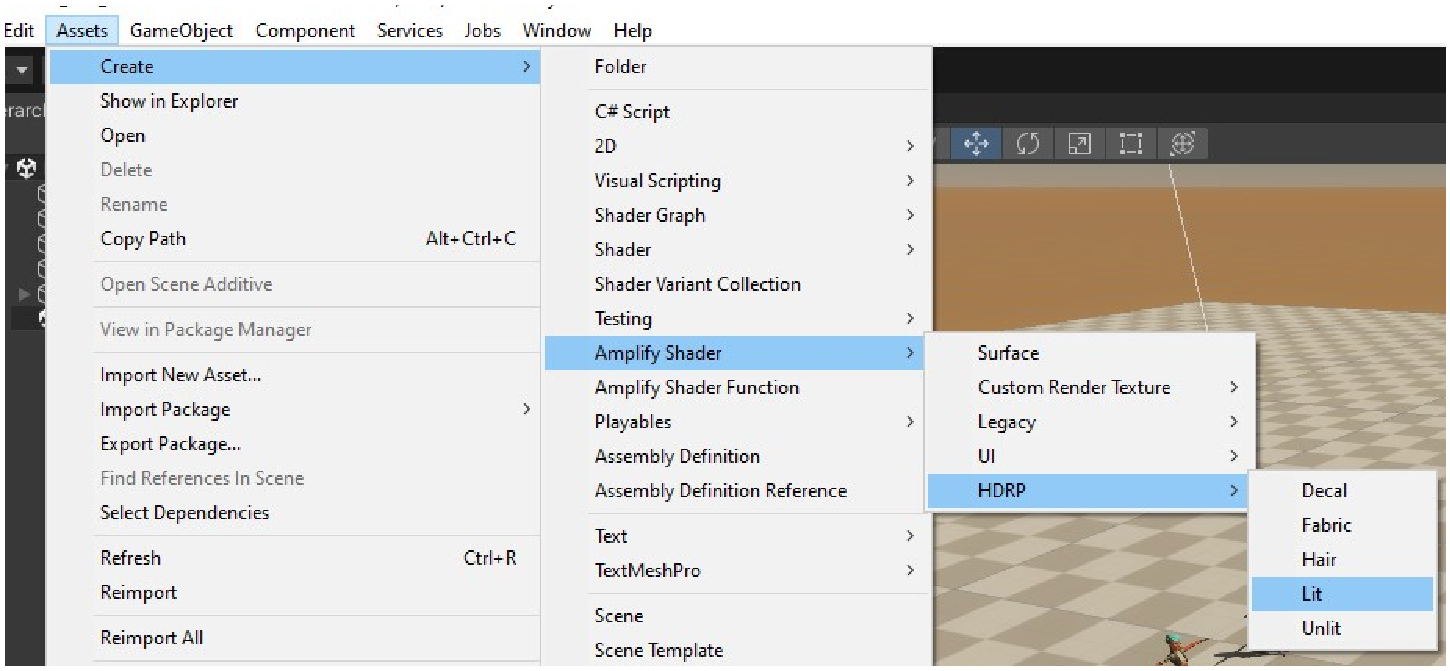


4. Save and assign this newly created shader to the materials of skinned mesh renderers.

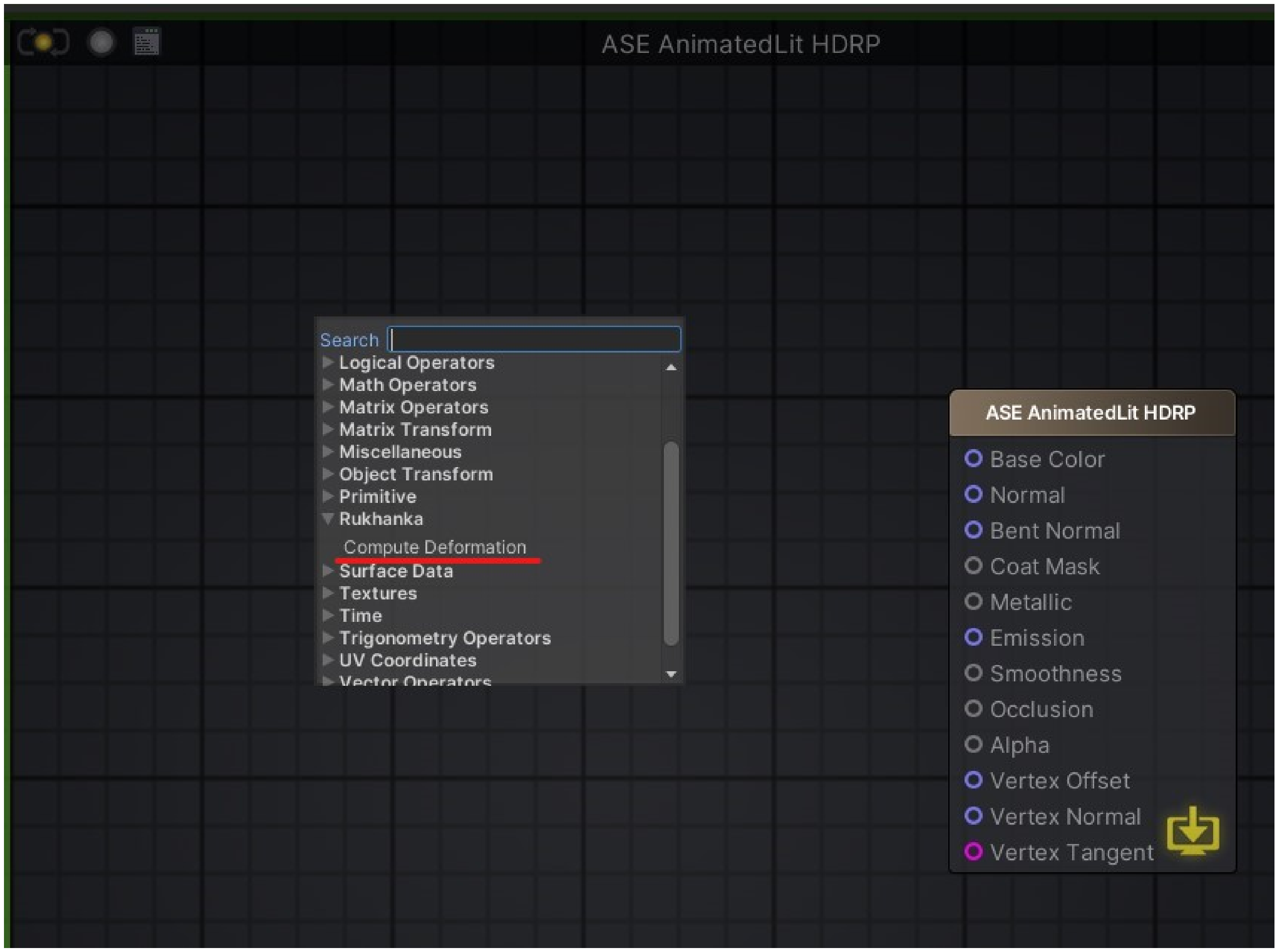
Amplify Shader Editor

`Amplify Shader Editor` tool has no `Entities.Graphics` compute deformation support out of the box, but **Rukhanka** adds necessary functionality to it. The process of creating deformation aware shader in `Amplify Shader Editor` is also simple:

1. Create `Amplify Shader` and open it for editing.



2. Open the `Add Node` dialog (`Space` or `Right Click`) and add the `Rukhanka`->`Compute Deformation` node.



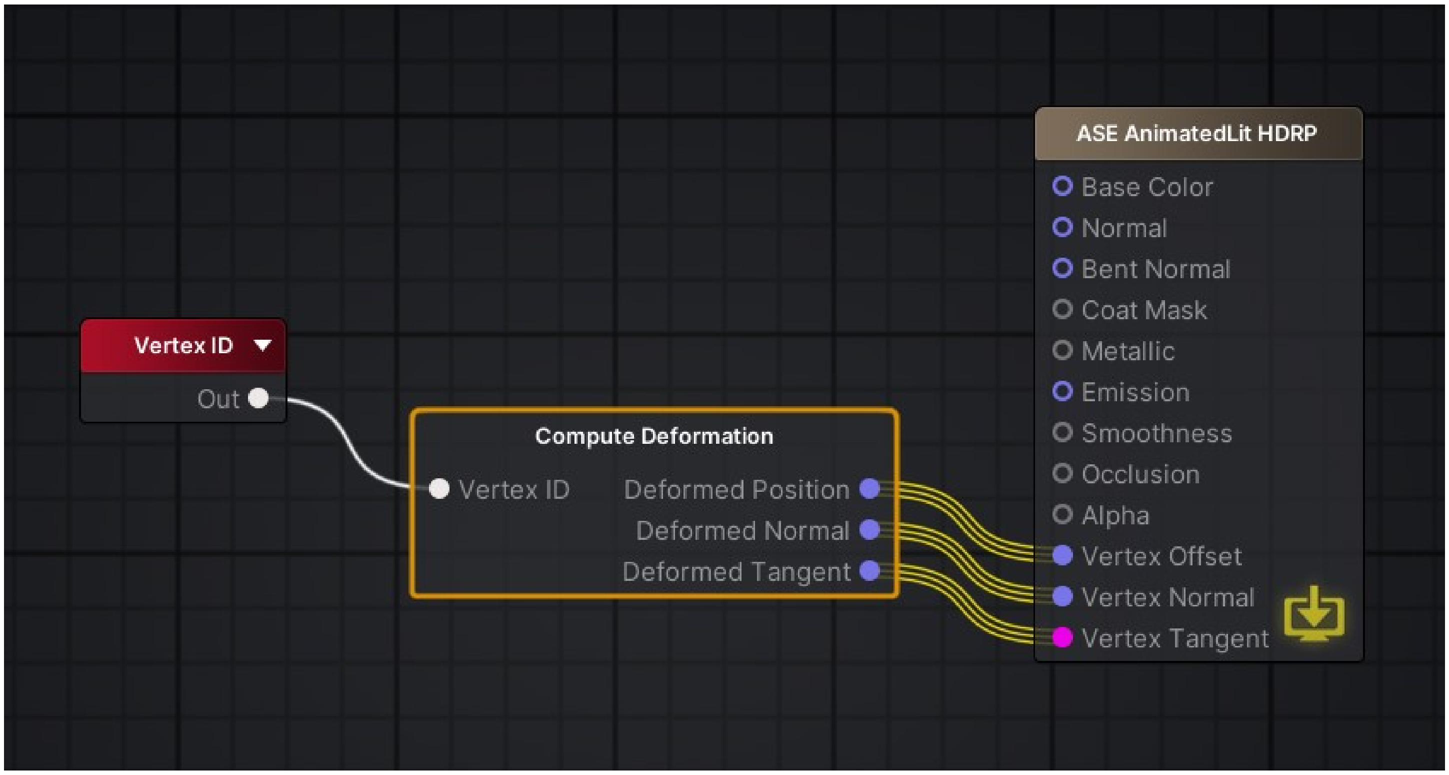
3. Add `Vertex Data`->`Vertex ID` node.

4. Connect the `Out` port of the `Vertex ID` node with the `Vertex ID` port of the `Compute Deformation` node.

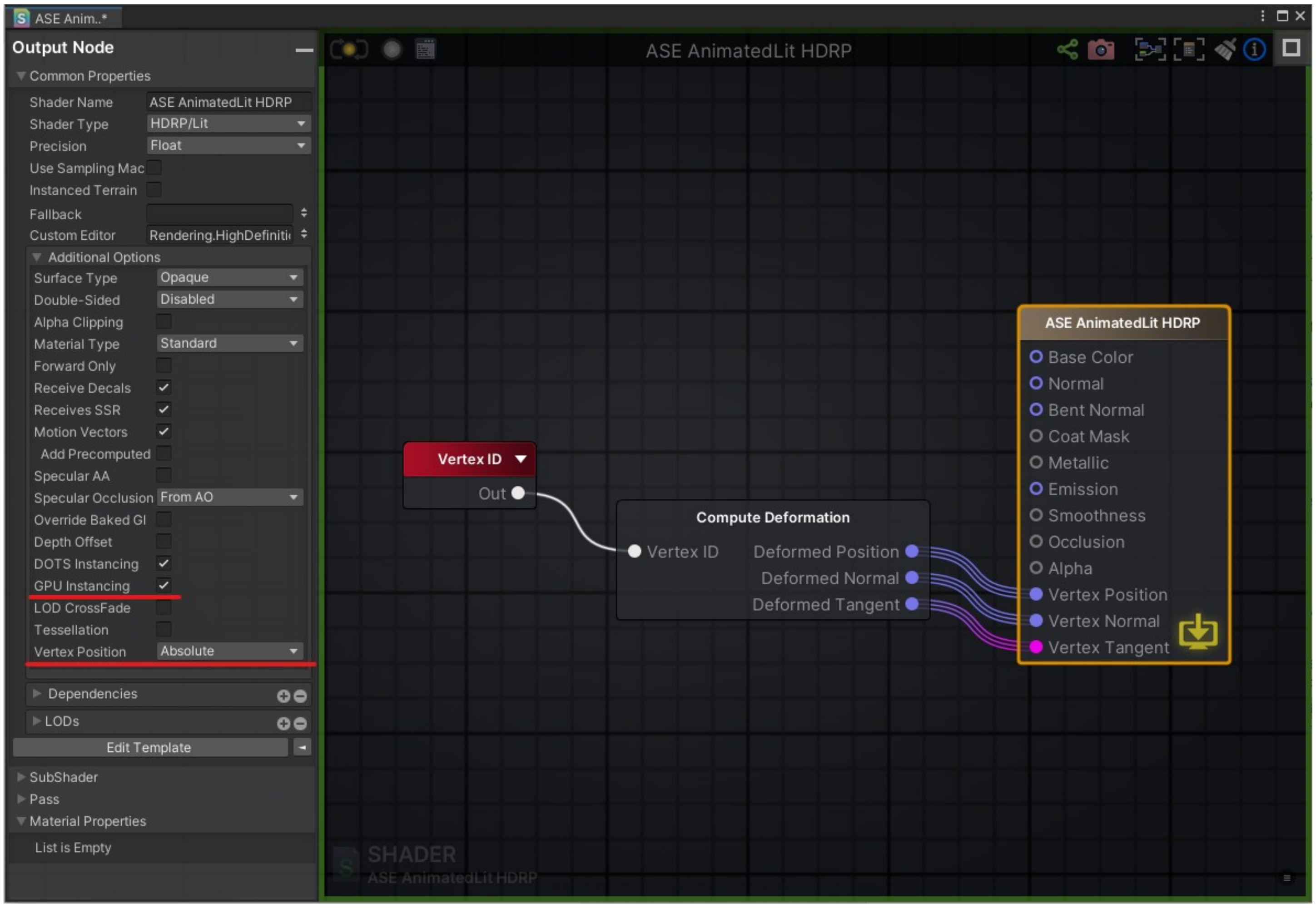
5. Connect the `Deformed Position` port of the `Compute Deformation` node with the `Vertex Offset` port of the shader master node.

6. Connect the `Deformed Normal` port of the `Compute Deformation` node with the `Vertex Normal` port of the shader master node.

7. Connect the `Deformed Tangent` port of the `Compute Deformation` node with the `Vertex Tangent` port of the shader master node.



8. Select the shader master node and enable the `DOTS Instancing` option, and set `Vertex Position` to `Absolute`.


ASE Anim..*

Output Node

Common Properties

Shader Name: ASE AnimatedLit HDRP
Shader Type: HDRP/Lit
Precision: Float
Use Sampling Mac: ☐
Instanced Terrain: ☐
Fallback:
Custom Editor: Rendering.HighDefiniti

Additional Options

Surface Type: Opaque
Double-Sided: Disabled
Alpha Clipping: ☐
Material Type: Standard
Forward Only: ☐
Receive Decals: ✓
Receives SSR: ✓
Motion Vectors: ✓
Add Precomputed: ☐
Specular AA: ☐
Specular Occlusion: From AO
Override Baked GI: ☐
Depth Offset: ☐
DOTS Instancing: ✓
GPU Instancing: ✓
LOD CrossFade: ☐
Tessellation: ☐
Vertex Position: Absolute

Dependencies

LODs

Edit Template

SubShader

Pass

Material Properties

List is Empty

Animator Parameters

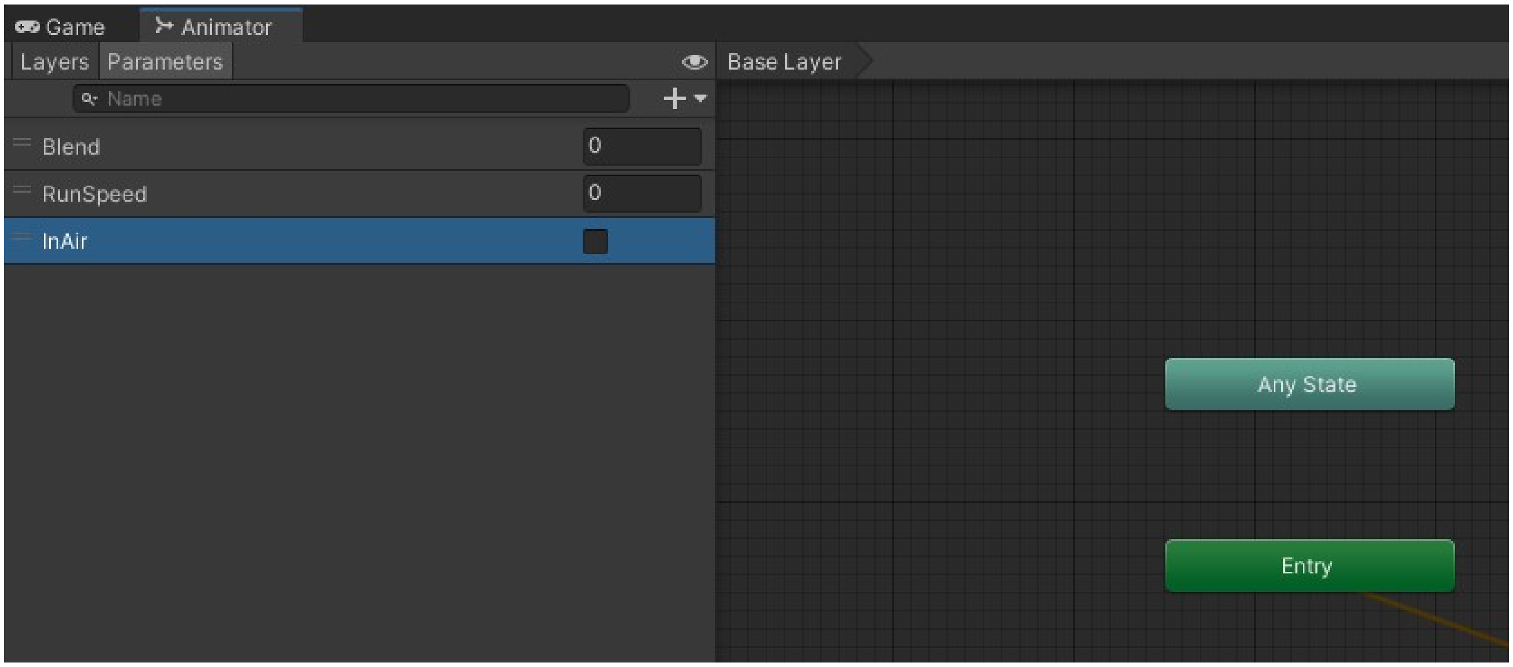
Direct Buffer Indexing

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 `FastAnimatorParameter`s 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
DynamicBuffer<AnimatorControllerParameterComponent> allParams)
    {
        var t = paramIndexTable.seedTable;

        blendParam.GetRuntimeParameterData(t, allParams, out var blendParamvalue);
        blendParam.SetRuntimeParameterData(t, allParams, new ParameterValue() {
floatValue = blendParamValue + 1 } );
        inAirParam.SetRuntimeParameterData(t, allParams, new ParameterValue() {
boolValue = true } );
    }
}
```

Animator Parameters Aspect

To further simplify animator parameter access there is `AnimatorParametersAspect` is introduced. It has several methods for convenient parameter data access:

```
[BurstCompile]
partial struct ProcessInputJob: IJobEntity
{
    public InputStateData inputData;

    public FastAnimatorParameter floatParam;
    public FastAnimatorParameter intParam;
    public FastAnimatorParameter triggerParam;
    public FastAnimatorParameter boolParam;

    void Execute(ref AnimatorParametersAspect paramAspect)
    {
        paramAspect.SetParameterValue(floatParam, 2.2f);
        paramAspect.SetParameterValue(intParam, 42);
        paramAspect.SetParameterValue(boolParam, true);
        paramAspect.SetTrigger(triggerParam);

        var floatValue = paramAspect.GetFloatParameter(floatParam);
        var boolValue = paramAspect.GetBoolParameter(boolParam);
    }
}
```

Some functions of `AnimatorParametersAspect` accept `FixedString` with parameter names. Those variants are slower than with `FastAnimatorParameter` and created mostly for easiness of quick prototyping and should not be used in final high performance code.

Entity Components

Rukhanka conceptually consists of two main modules:

- Animator controller
- Animation processor

Each module has its own baker system that prepares data for it by converting appropriate authoring components (Unity Animator, Unity Skinned Mesh Renderer, and Unity Animations)

Animator Controller System

The main function of the controller is advancing the animation state machine with time and preparing required animations for the current state and transitions. The animator controller system processes entities with the `AnimatorControllerLayerComponent` component array. Each element in this array represents separate animation layer as specified in Unity Animator.

```
public struct AnimatorControllerLayerComponent: IBufferElementData, IEnableableComponent
{
    ...
}
```

`AnimatorControllerLayerComponent` is inherit IEnableableComponent so can be enabled and disabled. If disabled, the state machine of this owning entity will not be processed, and the model stops its animations (pose will be paused). After enabling the state machine will continue from the moment of pause.

During state machine processing, all prepared animations will be arranged in form of an array of `AnimationToProcessComponent` components.

```
public struct AnimationToProcessComponent: IBufferElementData
{
    ...
}
```

Animation Process System

This system reads the `AnimationToProcessComponent` array, samples animations at specified times and blends results according to required blend rules. The animated entity is defined as `RigDefinitionComponent`:

```
public struct RigDefinitionComponent: IComponentData, IEnableableComponent
{
    ...
}
```

This component is also inherited from `IEnableableComponent` and can be enabled or disabled accordingly. In disabled state, all animations for an entity are not processed, but the corresponding state machine will continue its work and still provides the rig with updated animation data. After enabling `RigDefinitionComponent`, animations will jump to the actual state machine animation state.

Animator State Query

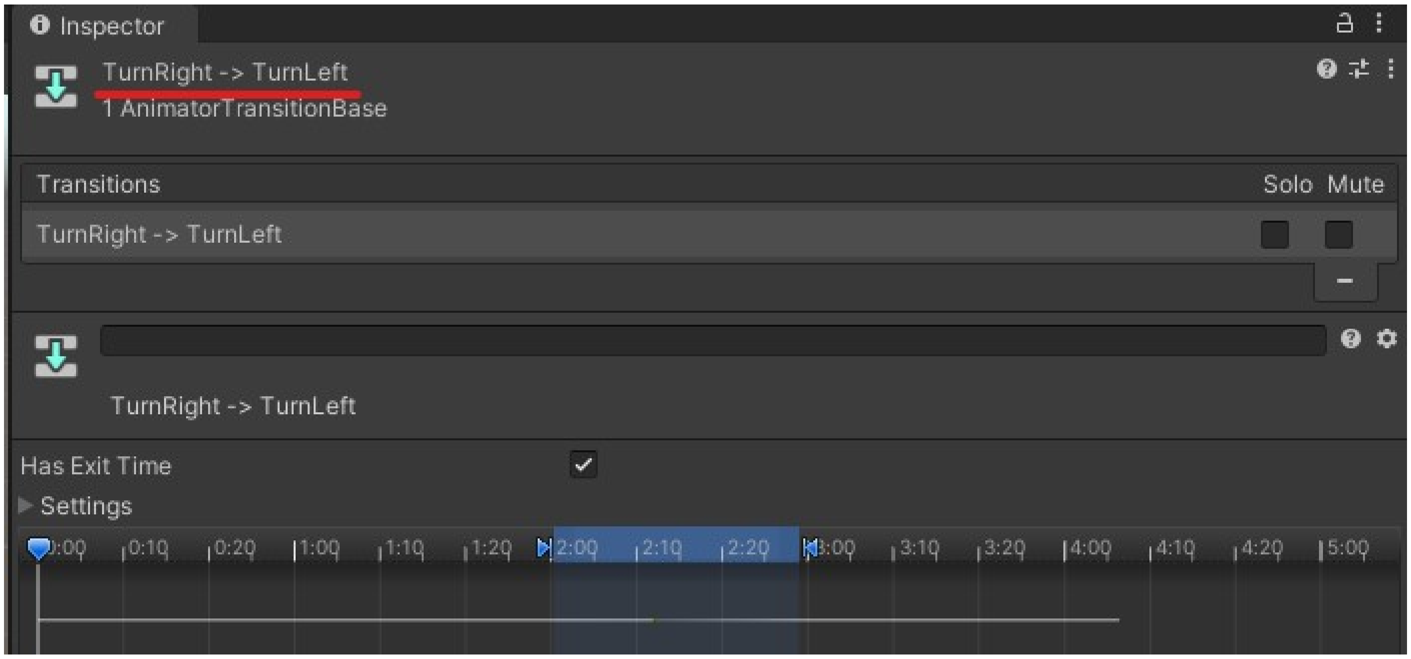
It is possible to query the runtime animator's internal state using the `AnimatorStateQueryAspect` aspect. State and transition data returned by this aspect contain a hash of state/transition respectively and normalized time (how much time controller is in this state/transition). To find out the required state, the hash value of it must be calculated upfront. This should be done by constructing `FixedStringName` with a state name. For example, a state with the name "Idle Random" can be used as follows:

```
var stateNameFull = new FixedStringName("Idle Random");
```

Hash code can be obtained from this string by calling `CalculateHash32()` member function:

```
var stateHash = stateNameFull.CalculateHash32();
```

For transitions, the process is the same:



```
var transitionName = new FixedStringName("Turn Right -> Turn Left");
var transitionHash = transitionName.CalculateHash32();
```

Here is a usage example of `AnimatorStateQueryAspect`:

```
public partial struct MySystem : ISystem
{
    uint myStateHash, myTransitionHash;

    public void OnStart(ref SystemState state)
    {
        myStateHash = new FixedStringName("State Name").CalculateHash32();
        myTransitionHash = new FixedStringName("State Name -> State Other
Name").CalculateHash32();
    }

    public void OnUpdate(ref SystemState state)
    {
        foreach (var animatorState in SystemAPI.Query<AnimatorStateQueryAspect>())
        {
            //  Specify required layer index of animator.
            var layerIndex = 0;

            //  Get animator current state
            var runtimeState = animatorState.GetLayerCurrentStateInfo(layerIndex);

            //  Use received RuntimeStateInfo structure to access to the current state
hash, and normalized state time
            if (runtimeState.hash == myStateHash)
            {
                //  Do something...
            }

            //  Get current transition
            var transitionState = animatorState.GetLayerCurrentTransitionInfo(layerIndex);
            if (myTransitionHash === transitionState.hash)
            {
                //  Do something...
            }
        }
    }
}
```

If the `RUKHANKA_DEBUG_INFO` script symbol is defined in the project, `RuntimeTransitionInfo` and `RuntimeStateInfo` structures will contain a `name` field with a transition/state symbolic name in it. There is an example named `Animator State Query` in **Rukhanka** samples. It shows described above features.

Non-skinned Mesh Animation

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

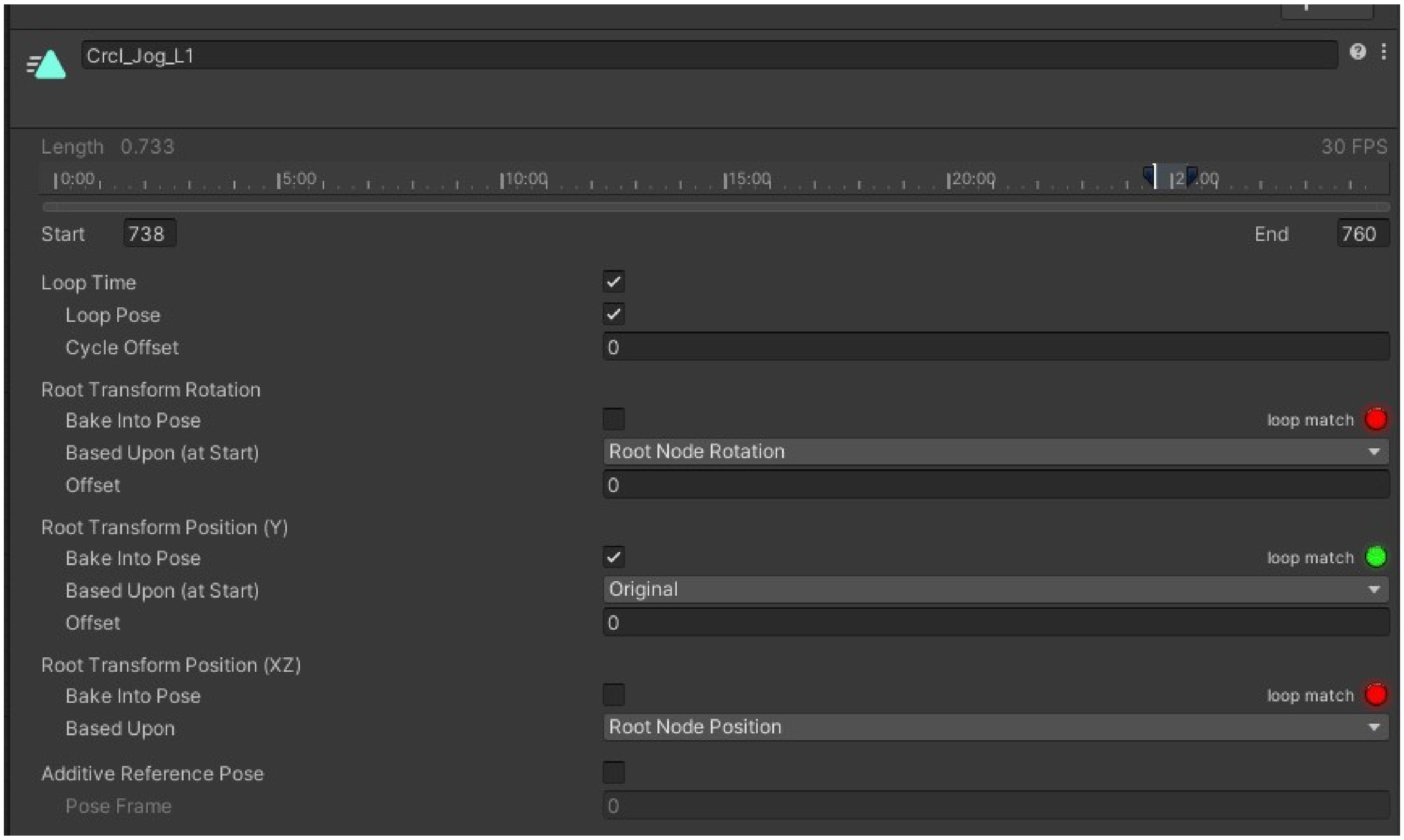
- Create an `Animator` and define animations in it.
- Place an object on subscene and add the `Rig Definition Authoring` Unity component to it.
- Add Unity `Animator` component, and assign the `Animator` state machine to it.
- **Rukhanka** will automatically create a rig for a given model starting from the node with `Rig Definition Authoring` assigned.

Root Motion

Rukhanka has full root motion support for humanoid and generic avatars and animations.

Follow these steps to enable root motion for your model:

- Configure `Root motion` as usual from motion configuration section of animator importer window:



Crcl_Jog_L1

Length 0.733 30 FPS

Start 738 End 760

Loop Time ✓
Loop Pose ✓
Cycle Offset 0

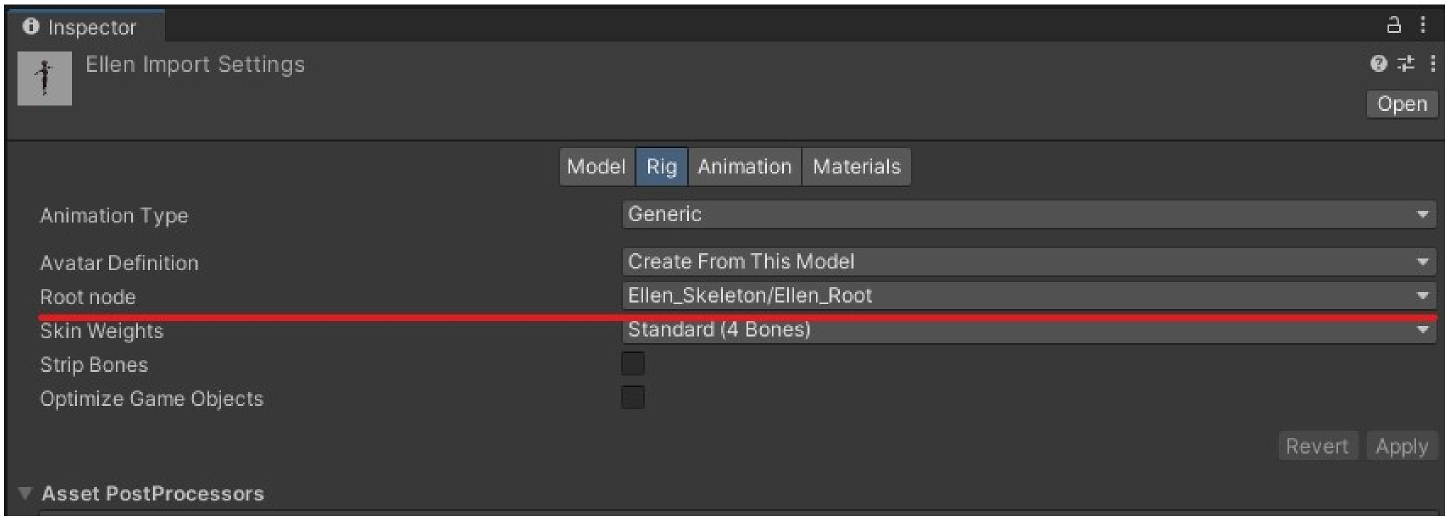
Root Transform Rotation
Bake Into Pose ☐ loop match 🔴
Based Upon (at Start) Root Node Rotation
Offset 0

Root Transform Position (Y)
Bake Into Pose ✓ loop match 🟢
Based Upon (at Start) Original
Offset 0

Root Transform Position (XZ)
Bake Into Pose ☐ loop match 🔴
Based Upon Root Node Position

Additive Reference Pose ☐
Pose Frame 0

- Configure `Root node` field in model importer window to the bone that will be used to drive object movement.


Inspector

Ellen Import Settings

Model | Rig | Animation | Materials

Animation Type: Generic

Avatar Definition: Create From This Model

Root node: Ellen_Skeleton/Ellen_Root

Skin Weights: Standard (4 Bones)

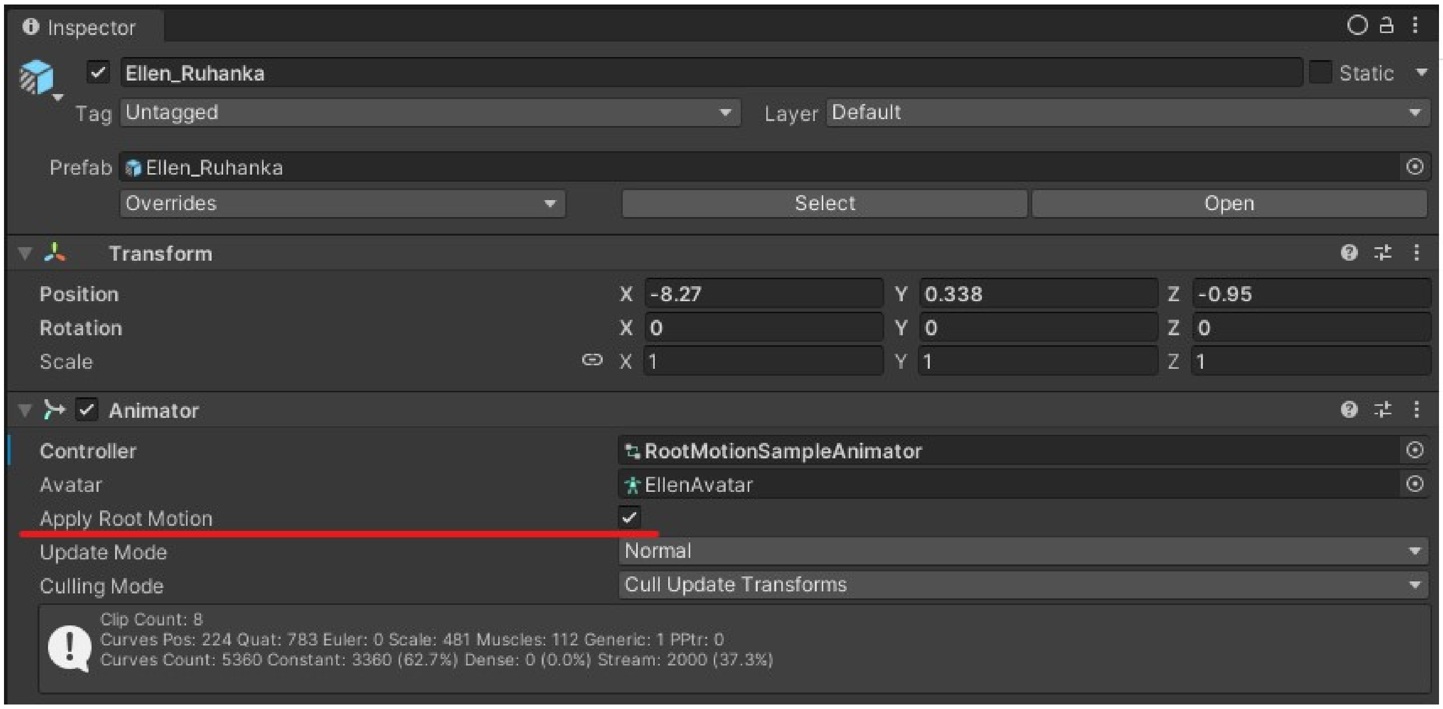
Strip Bones: ☐

Optimize Game Objects: ☐

Revert | Apply

Asset PostProcessors

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


Inspector

✓ Ellen_Ruhanka — Static

Tag: Untagged — Layer: Default

Prefab: Ellen_Ruhanka

Overrides | Select | Open

Transform

Position: X -8.27, Y 0.338, Z -0.95

Rotation: X 0, Y 0, Z 0

Scale: X 1, Y 1, Z 1

✓ Animator

Controller: RootMotionSampleAnimator

Avatar: EllenAvatar

Apply Root Motion: ✓

Update Mode: Normal

Culling Mode: Cull Update Transforms

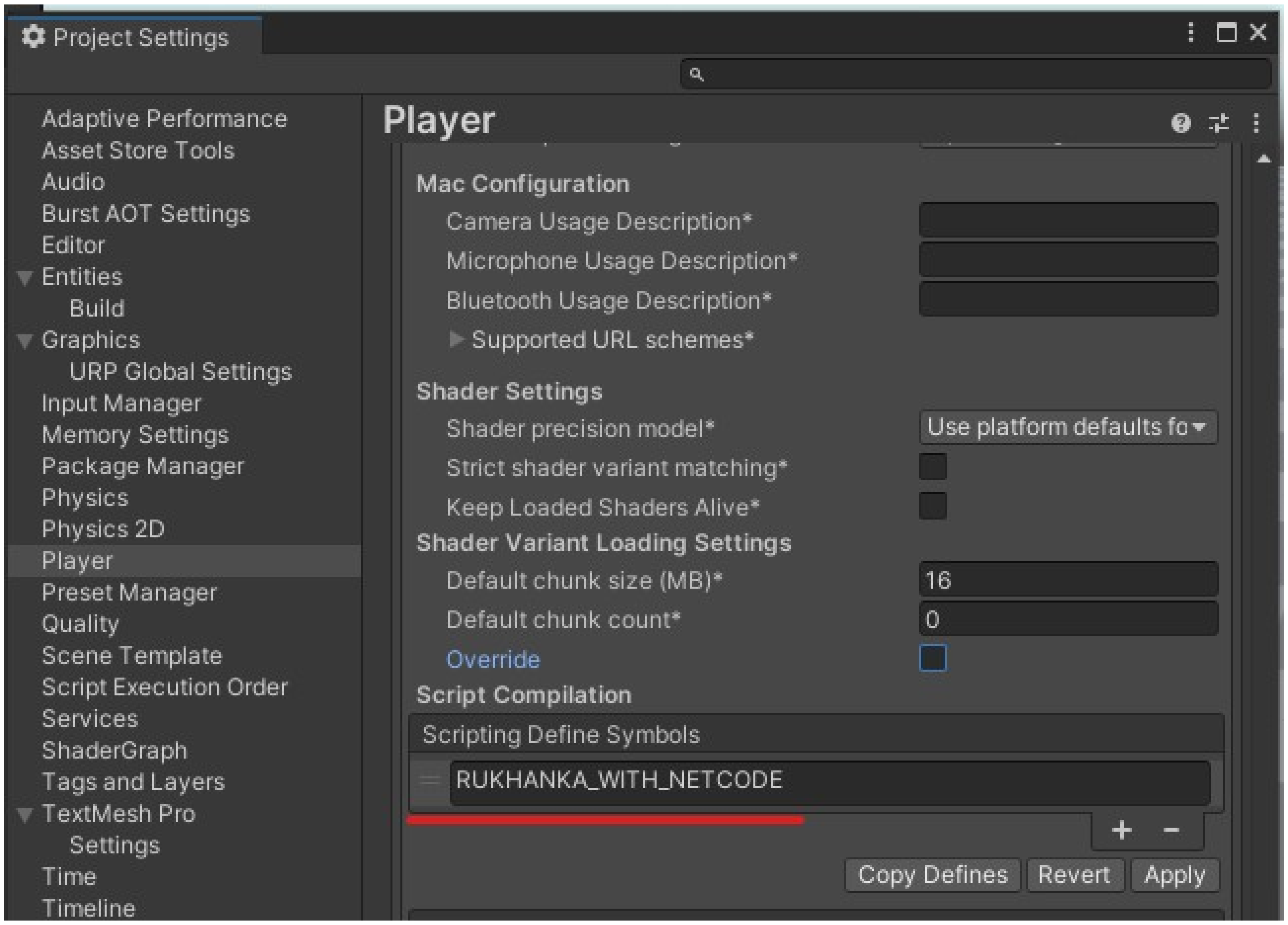
```
Clip Count: 8
Curves Pos: 224 Quat: 783 Euler: 0 Scale: 481 Muscles: 112 Generic: 1 PPtr: 0
Curves Count: 5360 Constant: 3360 (62.7%) Dense: 0 (0.0%) Stream: 2000 (37.3%)
```

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.

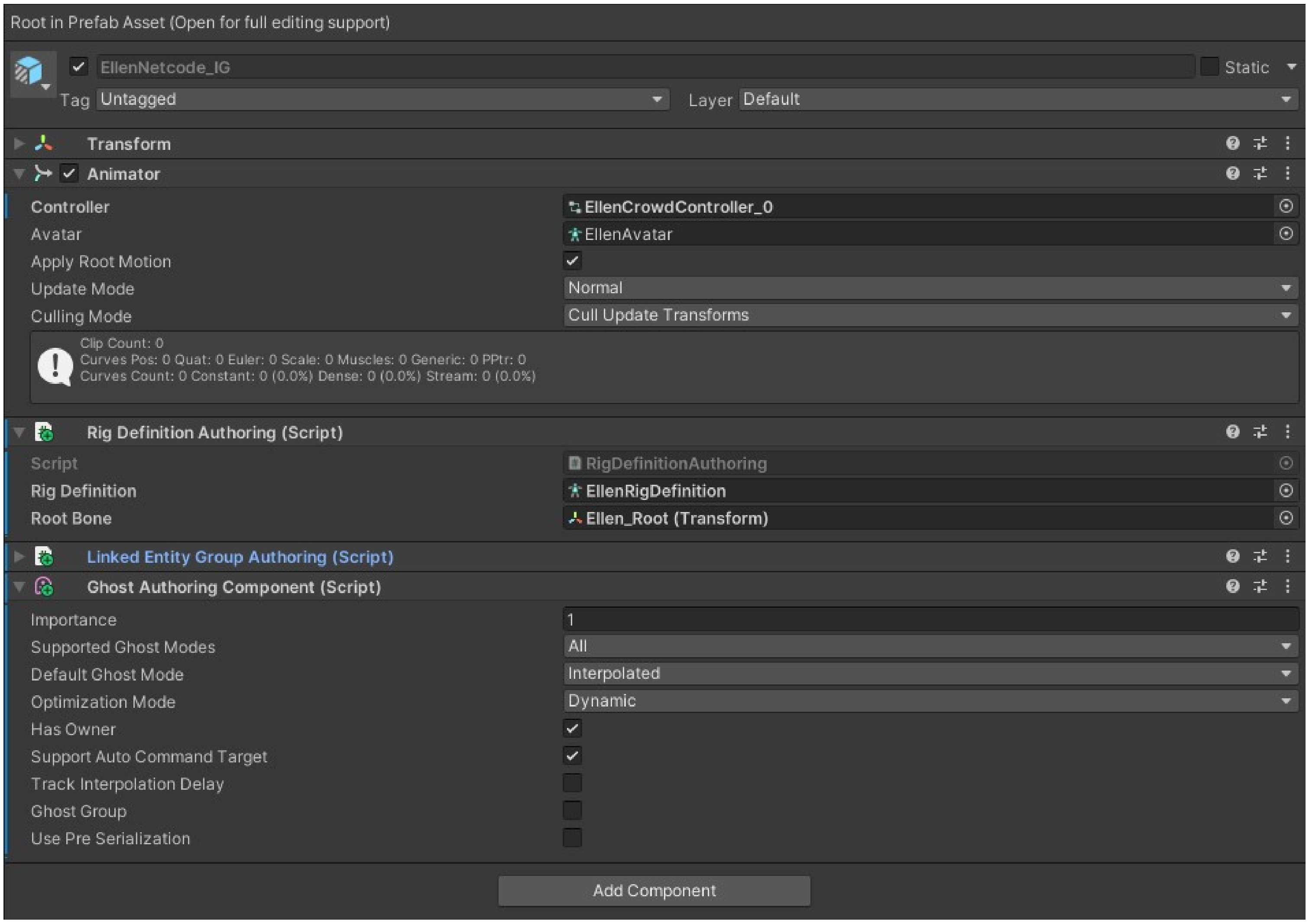
Working with Netcode

Rukhanka has full `Unity Netcode for Entities` package support. Network animation synchronization is available for predicted and interpolated ghosts. By default, **Rukhanka** is a client-only library. This means that it exists only in the client entity world. No data replication from server to clients is performed by the `Netcode` package. For configuring **Rukhanka** to be able to synchronize the state of animated entities over the network `RUKHANKA_WITH_NETCODE` script symbol should be defined in project settings.



Scripting Define Symbols: RUKHANKA_WITH_NETCODE

After that setup replicated prefab as described in the `Netcode` package documentation.


Root in Prefab Asset (Open for full editing support)

EllenNetcode_IG ✓ — Static
Tag Untagged — Layer Default

Transform

Animator ✓

Controller	EllenCrowdController_0
Avatar	EllenAvatar
Apply Root Motion	✓
Update Mode	Normal
Culling Mode	Cull Update Transforms

Clip Count: 0
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0 Generic: 0 PPtr: 0
Curves Count: 0 Constant: 0 (0.0%) Dense: 0 (0.0%) Stream: 0 (0.0%)

Rig Definition Authoring (Script)

Script	RigDefinitionAuthoring
Rig Definition	EllenRigDefinition
Root Bone	Ellen_Root (Transform)

Linked Entity Group Authoring (Script)

Ghost Authoring Component (Script)

Importance	1
Supported Ghost Modes	All
Default Ghost Mode	Interpolated
Optimization Mode	Dynamic
Has Owner	✓
Support Auto Command Target	✓
Track Interpolation Delay	
Ghost Group	
Use Pre Serialization	

Both types of ghost modes are supported. For predicted ghosts, there is a prediction version of `AnimatorControllerSystem` running in `Predicted Simulation System Group`. For interpolated ghosts animation data received from a server is used as is in animation calculation. Client-only animated entities (entities that do not require synchronization) will work as usual.

There is a special demo made for **Rukhanka** Netcode features showcase.

Working with Physics

Attached Physics Bodies

Physics bodies get unparented by `Unity.Physics` to work properly. **Rukhanka** will correctly animate an unparented bone entity.

> ⚠ **IMPORTANT**
>
> It is not immediately obvious, that a rigid body should be constructed in bone entity and not as child object of bone entity. In the latter case child entity will be unparented and lose connection with the animated bone. Please refer to the `Simple Physics` sample scene for setup example.

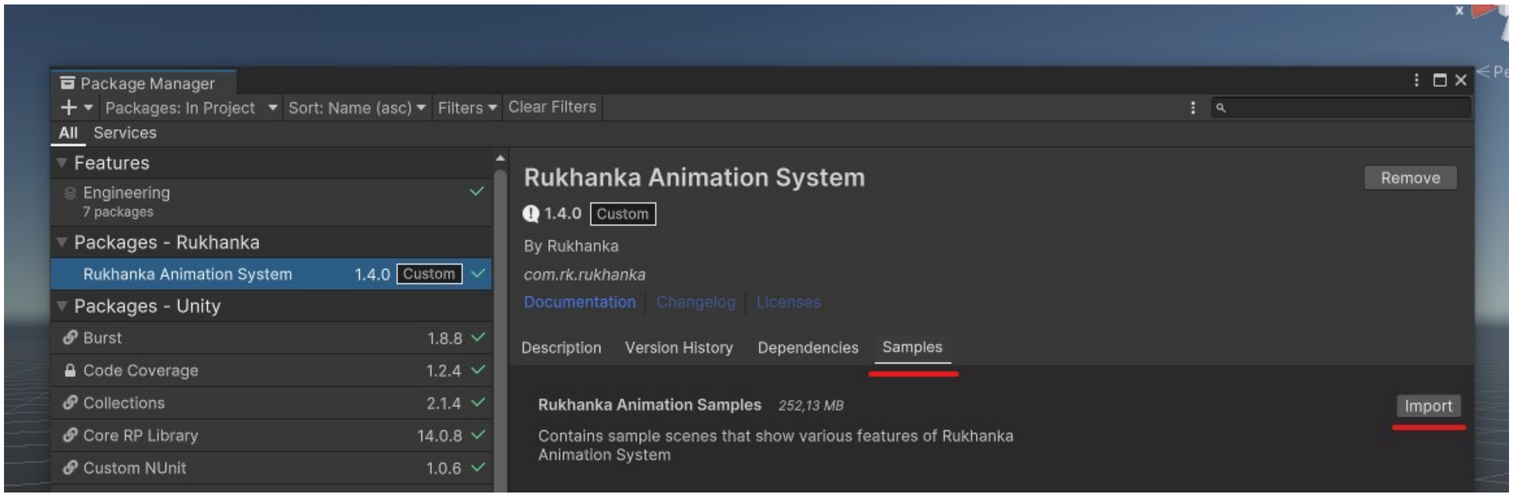
Ragdoll

Rig bone without animation will be driven by its `LocalTransform` component. This allows us to make a setup where bodies will drive animated bone positions and orientations. The `Ragdoll` sample scene will show this behavior.

Samples

Installation

Rukhanka's `Package Manager` description window has samples tab. Clicking on `Import` button will add **Rukhanka** samples in project folder under *Assets/Rukhanka Animation System/<version>/*. Navigate to the scenes subfolder for individual 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 and 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

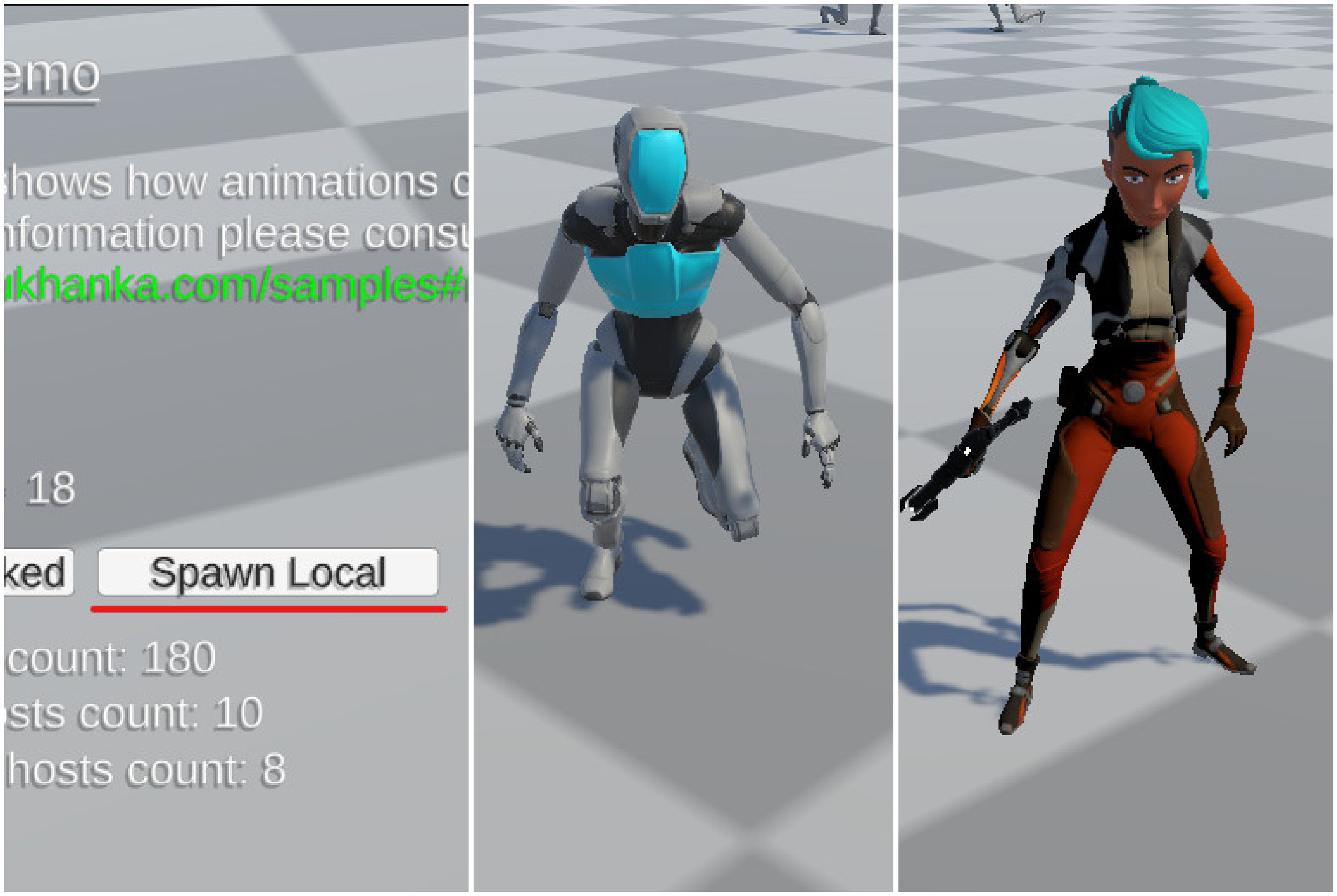
This sample is basically the same as the `Crowd` sample but with all skinned mesh models replaced by plain cubes. This step removes the big graphics pipeline pressure of the `Crowd` scene and keeps only raw **Rukhanka** animation system performance. This sample scene can be used for checking animation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

Netcode Demo

Netcode demo is made for showing **Rukhanka** ability to work with the `Unity Netcode for Entities` package to achieve client-server animation synchronization in a network game. The `RUKHANKA_WITH_NETCODE` script symbol should be defined for proper demo functionality.

Three types of objects that can be instantiated on the scene:

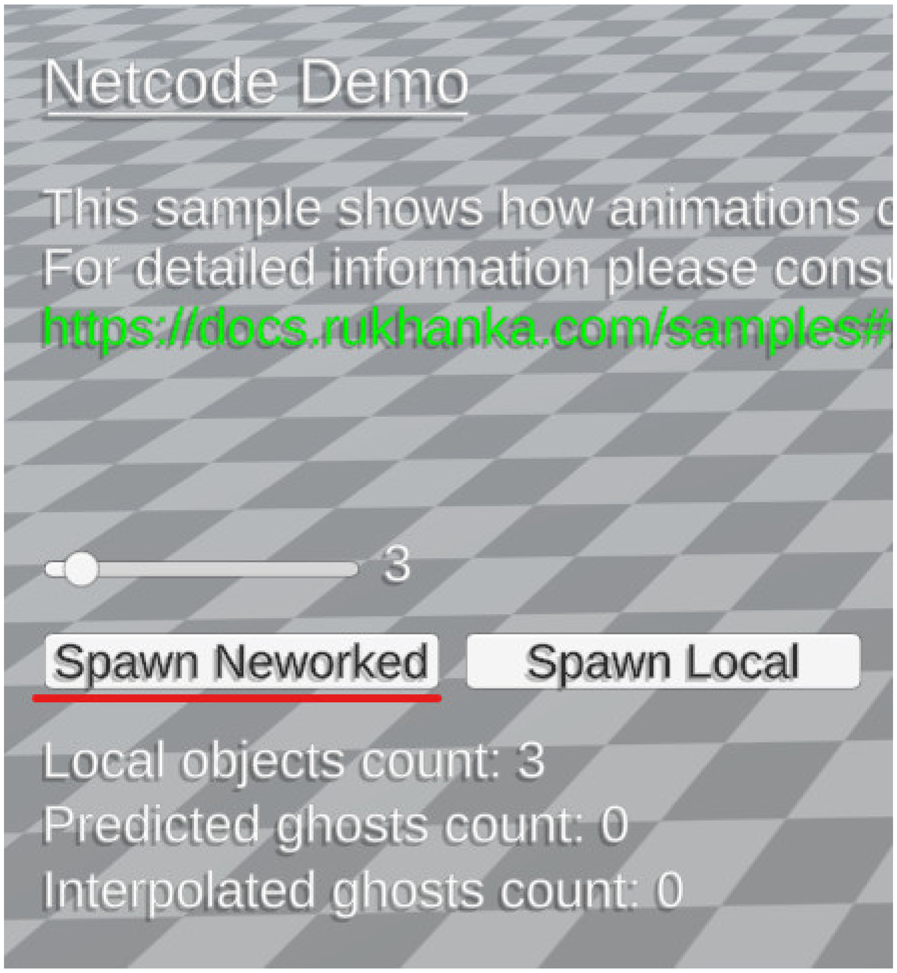
- Local client only. Those prefabs exist only in the client world and therefore not synchronized between client and server. To spawn such objects use the `Spawn Local` button. Client-only prefabs have a default coloring scheme to distinguish them:



- Interpolated ghosts. They are colored with red-tinted materials:

- Predicted ghosts. They are colored with green-tinted materials:

Both network synchronized prefab types can be created by pressing the `Spawn Networked` button:

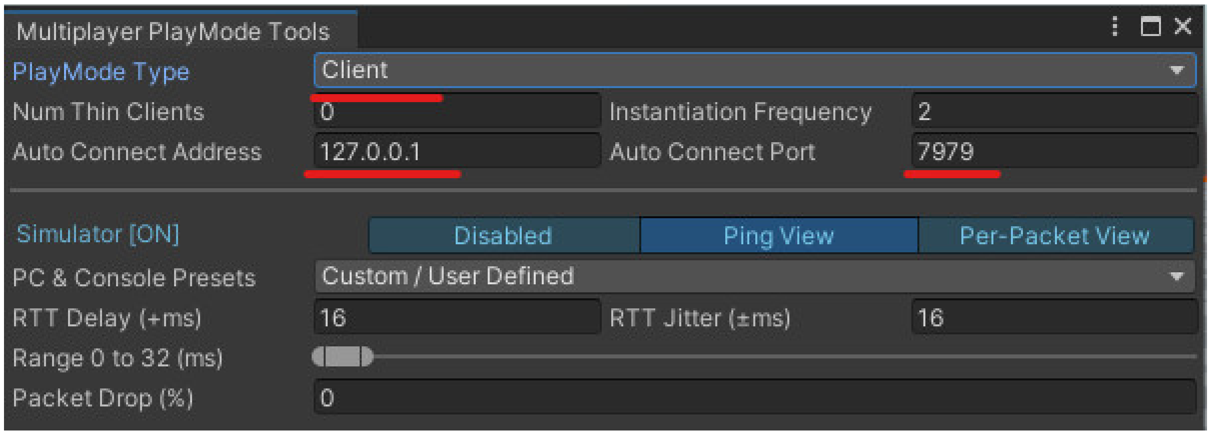

Netcode Demo

This sample shows how animations c
For detailed information please consu
https://docs.rukhanka.com/samples#

Local objects count: 3
Predicted ghosts count: 0
Interpolated ghosts count: 0

After this demo scene has been started both client and server worlds are created and the client automatically connects to the server. To observe interpolation and prediction behavior it is advised to use `PlayMode Tools` of the `Netcode` package. By simulating various packet loss and RTT conditions, differences between ghost modes can be observed in this sample.

To even better experience you can make a build with this scene, run it and then connect with another client instance directly from the editor by modifying the play mode type in `PlayMode Tools`. Use IP address 127.0.0.1 (localhost) and port 7979:


Multiplayer PlayMode Tools

PlayMode Type: Client
Num Thin Clients: 0
Instantiation Frequency: 2
Auto Connect Address: 127.0.0.1
Auto Connect Port: 7979

Simulator [ON]: Disabled | Ping View | Per-Packet View
PC & Console Presets: Custom / User Defined
RTT Delay (+ms): 16
RTT Jitter (±ms): 16
Range 0 to 32 (ms)
Packet Drop (%): 0

Using this test environment you can spawn networked prefabs from both of the clients and watch how they are replicated by the server.

Animator State Query

This sample shows the usage of the runtime animator query aspect. Every frame animator queried for its runtime state and transition and received information shown in scene UI.

Humanoid Animations

This sample shows humanoid avatar and animations usage. It has several controls that can be used to alter animation state machines and watch animation changes for sample models.

Simple Physics

Staring from v1.4.0 **Rukhanka** has an ability to work with unparented (flat) bone hierarchies. This allows to properly attach the `Unity.Physics` bodies to the **Rukhanka** skeleton bones. This sample shows described functionality.

Ragdoll

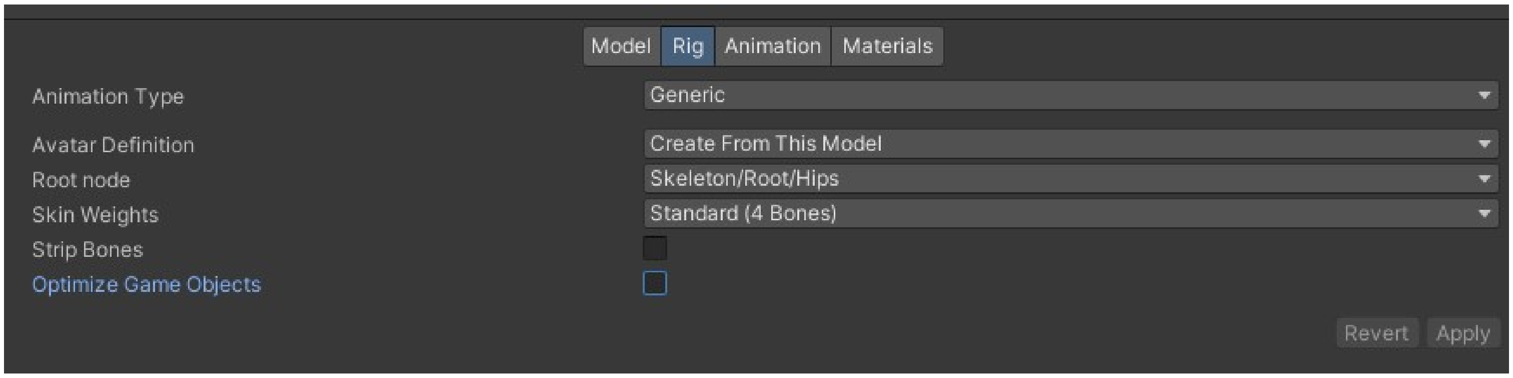
Bone entities can drive animated bone placement in animation rig. Ragdoll sample have very simple preconfigured physics ragdoll. When animations are not played for animated entity, physics simulation will drive animated bone positions.

Optimizing Rukhanka

Optimizing Bone Entities Count

By default, every animated skeleton bone has a corresponding entity in the ECS world. These entity positions get updated with computed animations by **Rukhanka**, and the hierarchy is processed by entity transformation systems. With a high bone count, updating these entities will take a significant amount of processing time. Often these bone entities are not needed, at least not all of them. It is advisable to keep only bones that are required for gameplay (bones with attachments, for example). **Rukhanka** provides functionality to strip unneeded bone entities.

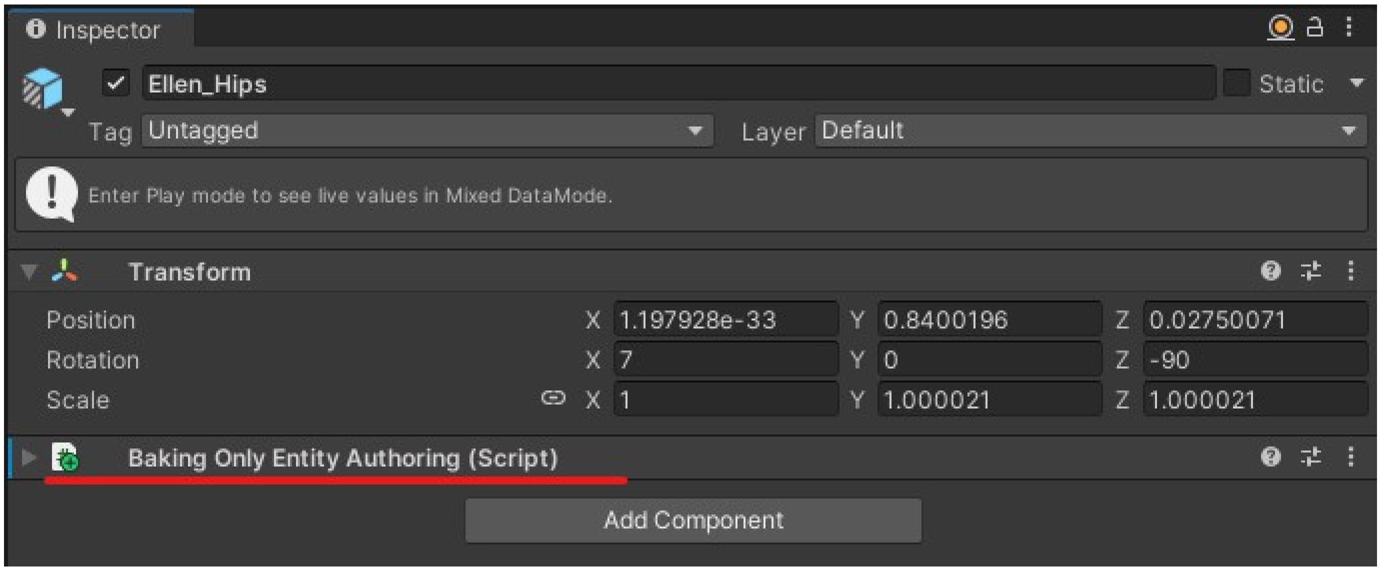
Unfortunately, Unity's builtin bone stripping functionality of the model importer window cannot be used:


Animation Type: Generic
Avatar Definition: Create From This Model
Root node: Skeleton/Root/Hips
Skin Weights: Standard (4 Bones)
Strip Bones: [ ]
Optimize Game Objects: [ ]

Rukhanka reads full hierarchy information from Unity avatar during the baking phase, so both `Strip Bones` and `Optimize Game Objects` checkboxes should be **unchecked**.

To remove unneeded entities from the ECS world there are two options available:

- Use the `BakingOnlyEntityAuthoring` component provided by the Entities package. By adding this component, the entity corresponding to this GameObject and all of its children will be stripped from the ECS world. Unfortunately, there is no option to configure its behavior for keeping some of its children (often attachments are bound to the bone deeper in the hierarchy). This approach is very good for situations when the model does not have attachments at all. Adding the `BakingOnlyEntityAuthoring` component to the rig root bone will strip the entire bone entities hierarchy.

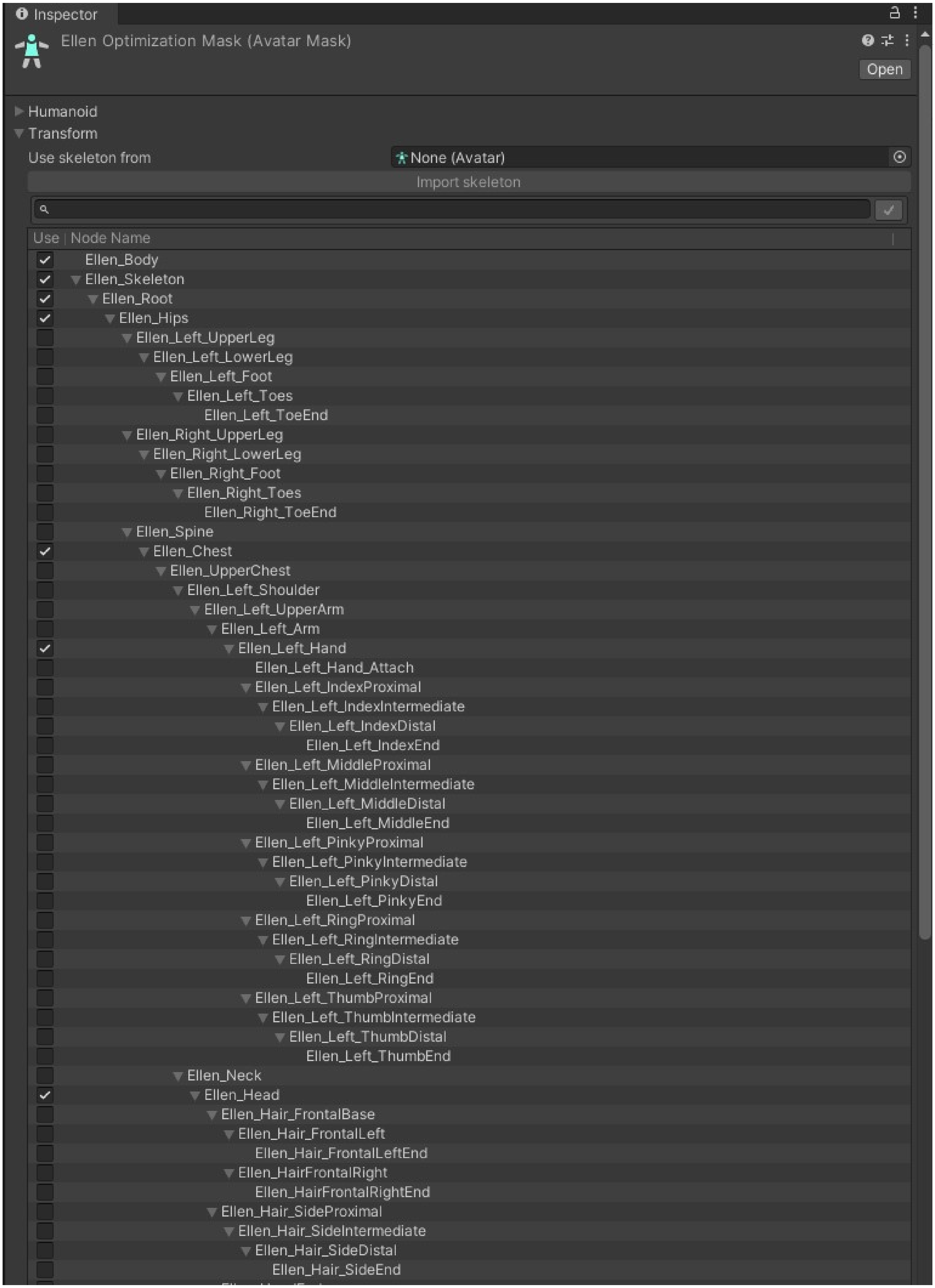


Enter Play mode to see live values in Mixed DataMode.

	X	Y	Z
Position	1.197928e-33	0.8400196	0.02750071
Rotation	7	0	-90
Scale	1	1.000021	1.000021

Baking Only Entity Authoring (Script)

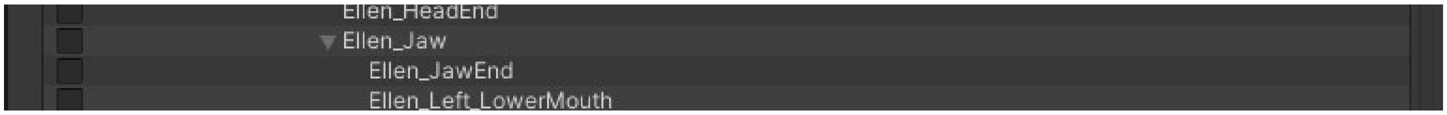
- If more precise stripping granularity is needed, then a special bone mask can be used. Optimization mask is just the default Unity AvatarMask object. Fill it with the specifying avatar in the `Use skeleton from` field and press the `Import skeleton` button. Enabled bones in it will stay as entities and disabled ones will be stripped:



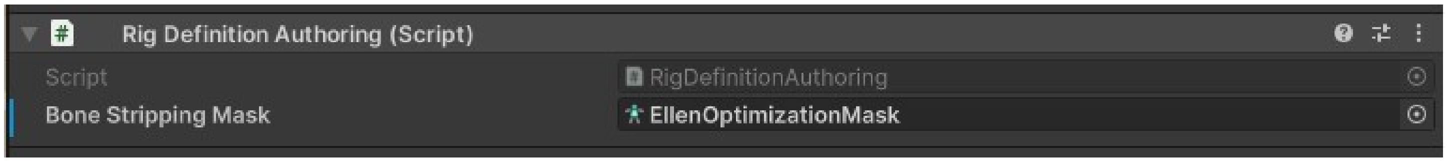
Ellen Optimization Mask (Avatar Mask)

Humanoid
Transform
Use skeleton from: None (Avatar)

Use	Node Name
✓	Ellen_Body
✓	Ellen_Skeleton
✓	Ellen_Root
✓	Ellen_Hips
	Ellen_Left_UpperLeg
	Ellen_Left_LowerLeg
	Ellen_Left_Foot
	Ellen_Left_Toes
	Ellen_Left_ToeEnd
	Ellen_Right_UpperLeg
	Ellen_Right_LowerLeg
	Ellen_Right_Foot
	Ellen_Right_Toes
	Ellen_Right_ToeEnd
	Ellen_Spine
✓	Ellen_Chest
	Ellen_UpperChest
	Ellen_Left_Shoulder
	Ellen_Left_UpperArm
	Ellen_Left_Arm
✓	Ellen_Left_Hand
	Ellen_Left_Hand_Attach
	Ellen_Left_IndexProximal
	Ellen_Left_IndexIntermediate
	Ellen_Left_IndexDistal
	Ellen_Left_IndexEnd
	Ellen_Left_MiddleProximal
	Ellen_Left_MiddleIntermediate
	Ellen_Left_MiddleDistal
	Ellen_Left_MiddleEnd
	Ellen_Left_PinkyProximal
	Ellen_Left_PinkyIntermediate
	Ellen_Left_PinkyDistal
	Ellen_Left_PinkyEnd
	Ellen_Left_RingProximal
	Ellen_Left_RingIntermediate
	Ellen_Left_RingDistal
	Ellen_Left_RingEnd
	Ellen_Left_ThumbProximal
	Ellen_Left_ThumbIntermediate
	Ellen_Left_ThumbDistal
	Ellen_Left_ThumbEnd
	Ellen_Neck
✓	Ellen_Head
	Ellen_Hair_FrontalBase
	Ellen_Hair_FrontalLeft
	Ellen_Hair_FrontalLeftEnd
	Ellen_HairFrontalRight
	Ellen_HairFrontalRightEnd
	Ellen_Hair_SideProximal
	Ellen_Hair_SideIntermediate
	Ellen_Hair_SideDistal
	Ellen_Hair_SideEnd


Ellen_HeadEnd
Ellen_Jaw
Ellen_JawEnd
Ellen_Left_LowerMouth

- Then set the configured avatar mask in `RigDefinitionAuthoring`:


Rig Definition Authoring (Script)
Script RigDefinitionAuthoring
Bone Stripping Mask EllenOptimizationMask

IMPORTANT

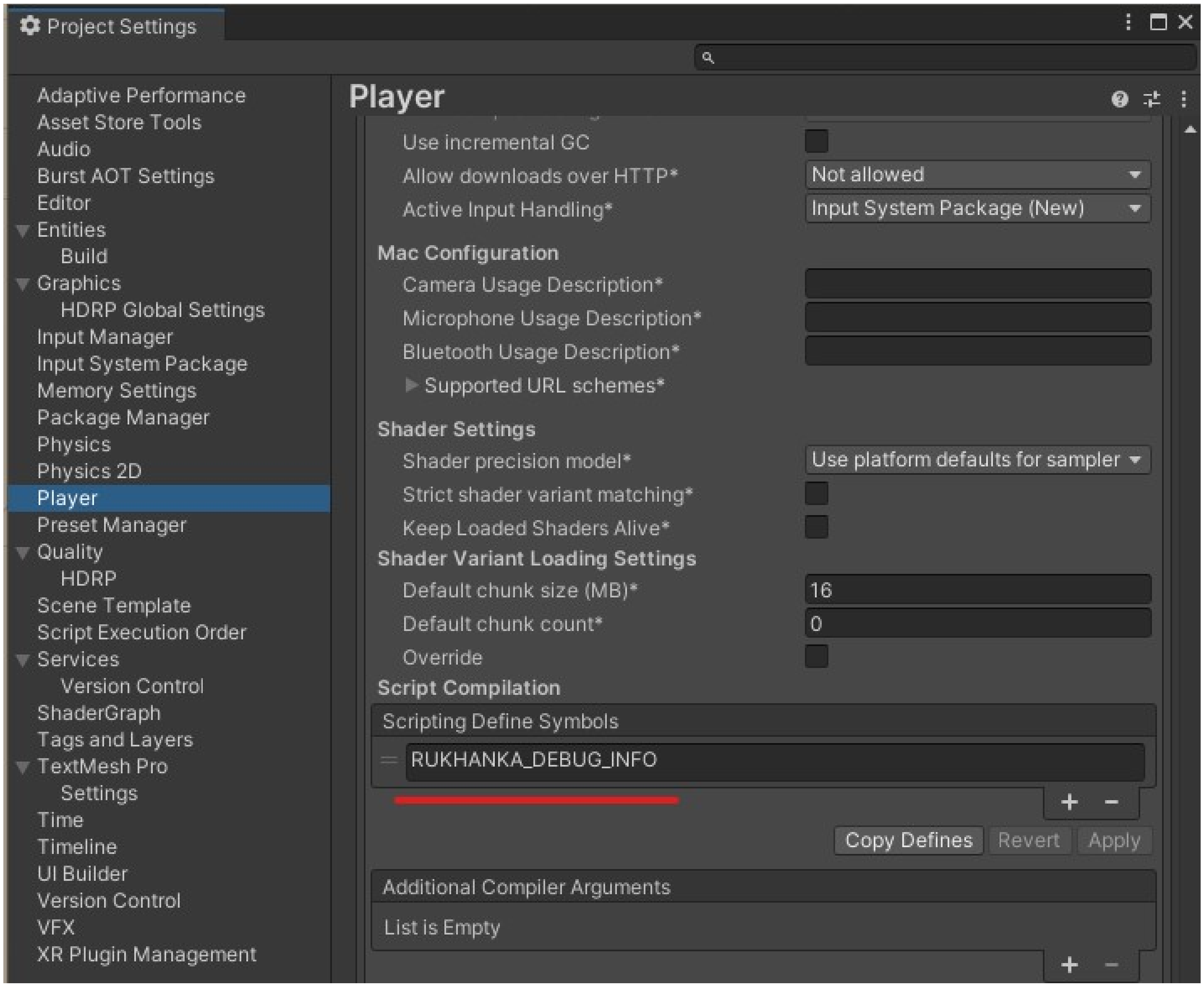
By using a bone stripping mask whole entity bone hierarchy will be flattened (i.e. `Parent` component will be removed from bone entities). Unparented bones must have a valid animation track (just one identity keyframe will suffice) associated with it to be animated correctly. Otherwise, the bone will be driven by `LocalTransform` component values, which is a world pose if the parent is not present.

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 `RUKHANKA_DEBUG_INFO` script definition symbol into project preferences:



Player

Use incremental GC

Allow downloads over HTTP* — Not allowed

Active Input Handling* — Input System Package (New)

Mac Configuration

Camera Usage Description*

Microphone Usage Description*

Bluetooth Usage Description*

Supported URL schemes*

Shader Settings

Shader precision model* — Use platform defaults for sampler

Strict shader variant matching*

Keep Loaded Shaders Alive*

Shader Variant Loading Settings

Default chunk size (MB)* — 16

Default chunk count* — 0

Override

Script Compilation

Scripting Define Symbols

RUKHANKA_DEBUG_INFO

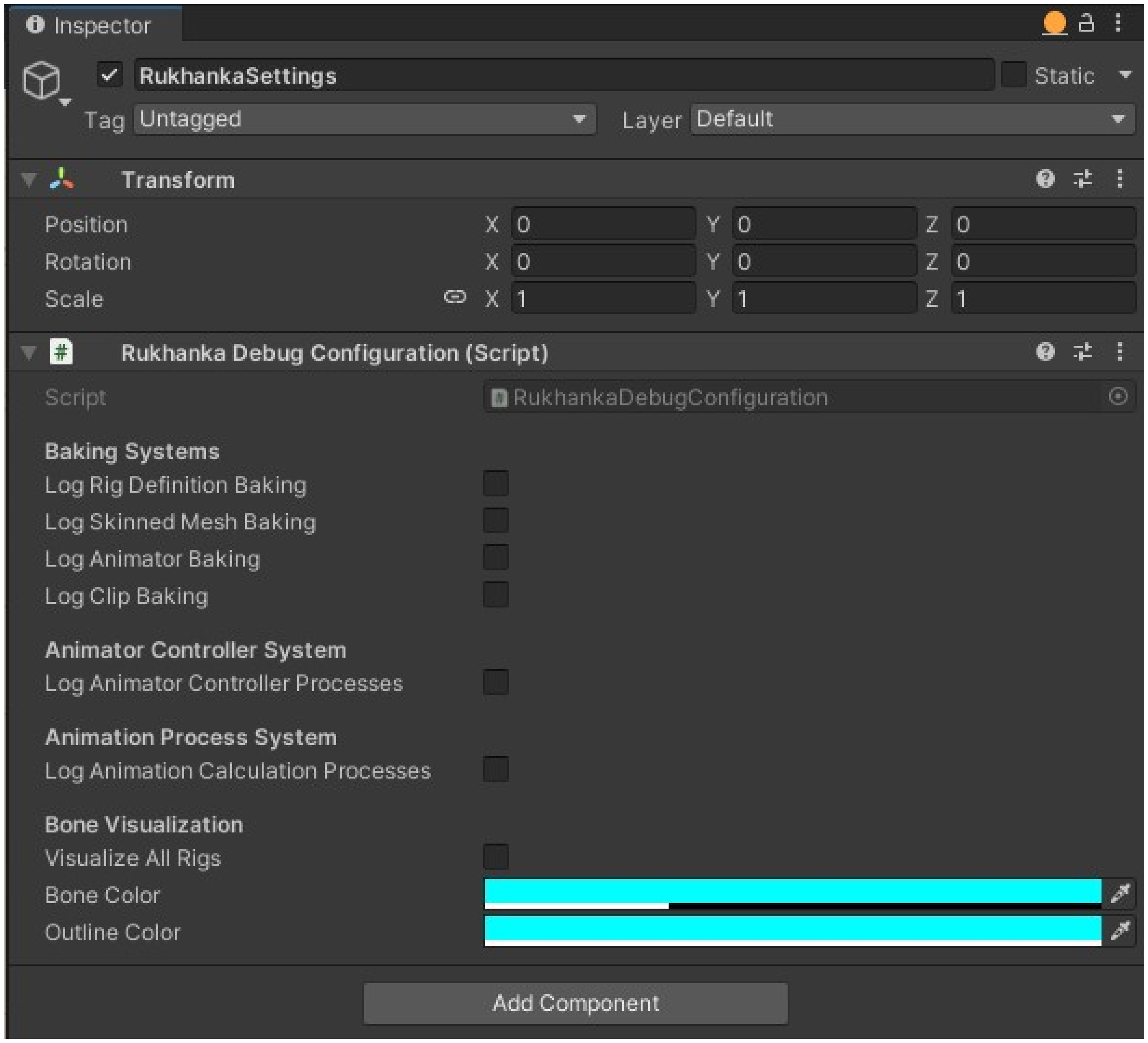
Additional Compiler Arguments

List is Empty

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 `RUKHANKA_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`.


Inspector

RukhankaSettings — Static
Tag: Untagged — Layer: Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Rukhanka Debug Configuration (Script)
Script: RukhankaDebugConfiguration

Baking Systems
Log Rig Definition Baking
Log Skinned Mesh Baking
Log Animator Baking
Log Clip Baking

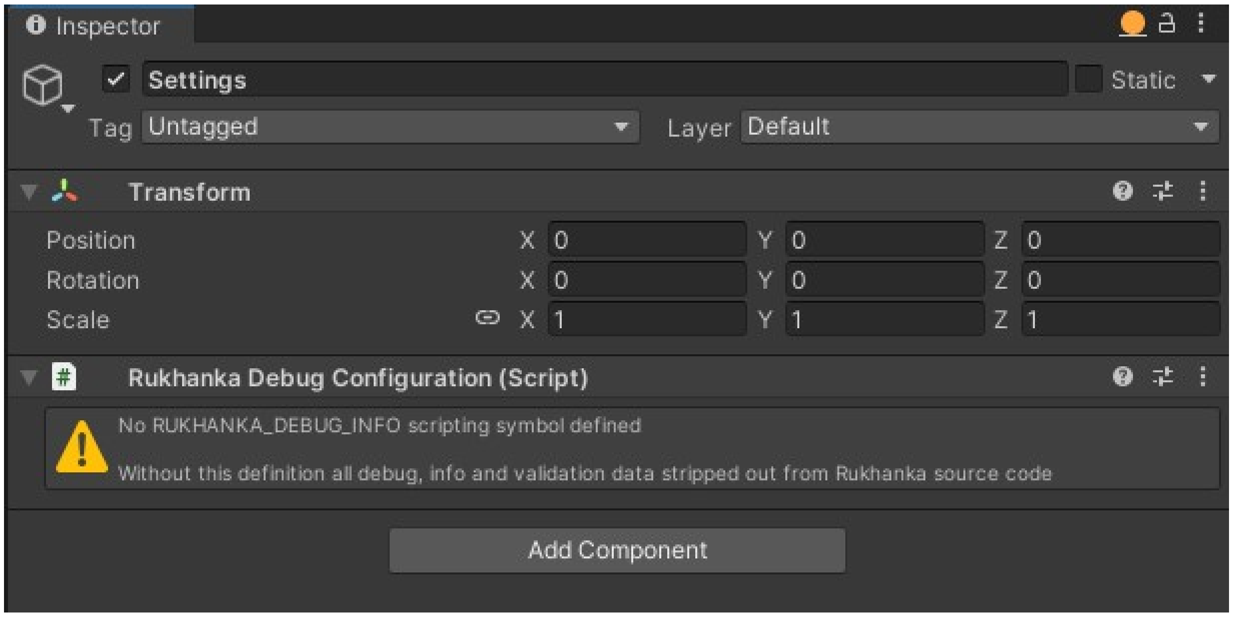
Animator Controller System
Log Animator Controller Processes

Animation Process System
Log Animation Calculation Processes

Bone Visualization
Visualize All Rigs
Bone Color
Outline Color

Add Component

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


Inspector

Settings — Static
Tag: Untagged — Layer: Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Rukhanka Debug Configuration (Script)

No RUKHANKA_DEBUG_INFO scripting symbol defined

Without this definition all debug, info and validation data stripped out from Rukhanka source code

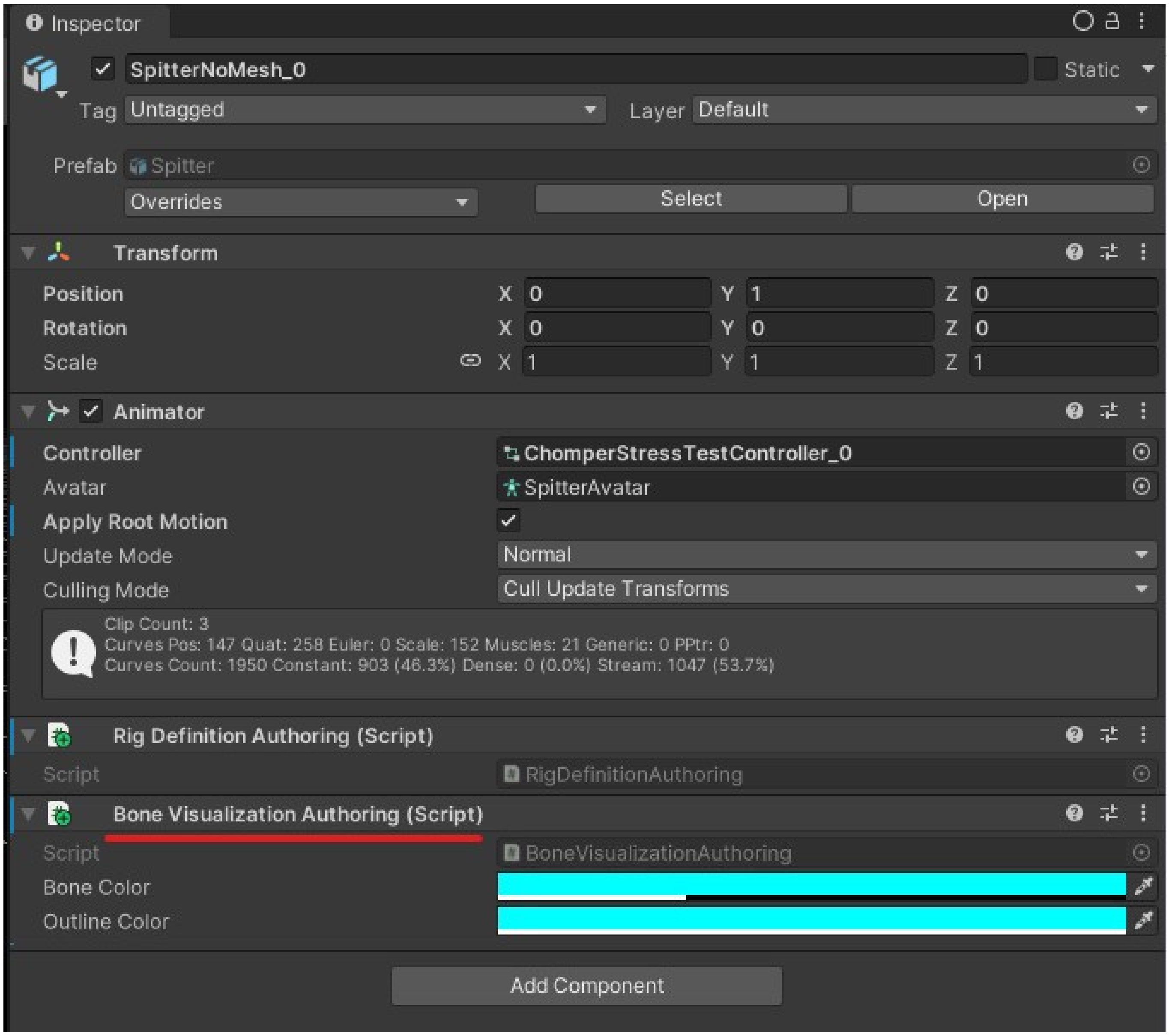
Add Component

- *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

Bone Visualization Authoring (Script)

Script BoneVisualizationAuthoring
Bone Color
Outline Color

Add Component

Changelog

[1.4.1]

Fixed

- All runtime systems `TempJob` allocations was changed to `WorldUpdateAllocator`.
- Code cleanup with more extensive `SystemAPI` usage.
- Moving shared utility code into separate `Rukhanka.Toolbox` assembly.
- Incorrect root bone indexing for humanoid rigs.
- Incorrect non-root motion of humanoid rig hips.

[1.4.0]

Fixed

- Several memory leaks during baking and runtime.
- Incorrect additive animation calculation for humanoid animations.

Added

- Two new sample scenes: `Ragdoll` and `Simple Physics`.
- Animation keyframe binary search was implemented.
- **Rukhanka** can now work with unparented bone entities. This allows to properly handle physics body bone attachments.
- Unneeded bone entity stripping functionality. Refer to documentation for the detailed description.
- Internal bone animation data is now exposed as a `RuntimeAnimationData` singleton.
- Full root motion support for `Humanoid` and `Generic` rigs. All root motion animation configuration parameters are supported.
- **Rukhanka** now requires `UNITY_BURST_EXPERIMENTAL_ATOMIC_INTRINSICS` script compilation symbol. It will be added to project scripting define symbols automatically if not present.

Changed

- Updated Entities dependency to version 1.0.16.
- Removed synchronization point related to `AnimationToProcess` buffer filling.
- Removed synchronization point related to root motion delta states processing.
- The animation process system is split into two distinct parts: animation calculation and animation application. This allows to injection of animation results post-processing and modifications (for example IK) functionality.
- Runtime created bone-name-to-index hash map was removed. It has been replaced with a blob perfect hash map created during baking time.
- Samples were moved to the samples tab of package properties. There are shared HDRP/URP sample scenes now. Materials and scene properties will adapt to the current renderer pipeline automatically.
- Animation controller layer weight is a runtime property of AnimatorControllerLayerComponent now.

[1.3.1]

Fixed

- Fixed compilation errors during standalone builds creation.

[1.3.0]

Added

- Humanoid-type support for models and animations.
- Humanoid avatar mask support.
- Humanoid avatars and animations sample scene.

Changed

- The rig definition authoring script now contains zero configuration fields. The avatar mask used for rig definition is not needed anymore. All required information **Rukhanka** reads from `Unity Avatar`. This is a breaking change. Please carefully read the upgrade process.

Fixed

- `AnimatorControllerParameterComponent` buffer did not replicate with NetCode.

[1.2.1]

Fixed

- Fixed compilation errors during standalone builds creation.

[1.2.0]

Added

- Compute deformation node for `Amplify Shader Editor`. Now it is possible to make `Entities.Graphics` deformation-compatible shaders with this tool.
- Trigger set API for `AnimatorParametersAspect`.
- The animator parameter access performance tests.
- Own entity command buffer system for optimizing ECB usage after `AnimationControllerSystem`.
- `AnimatorStateQuery` aspect for access to runtime animator data.

Changed

- Animator parameter internal hash code representation was moved from `Hash128` to uint. This leads to a smaller `AnimatorControllerParameterComponent` size and better chunk utilization.

Fixed

- The state machine states without an assigned motion field had incorrect weight calculations.
- Exit and enter transition events that happened in the same frame lead to one incorrectly processed frame. This was clearly observable with transitions from/to "no-motion" states.
- Trigger parameters were reset even if the transition cumulative condition (all conditions must be true) is not met.
- Entering through the sub-state machine's `Enter` state was handled incorrectly.

- Exiting from nested sub-state machines using the `Exit` state was handled incorrectly.
- Multiple transitions from the `Enter` state machine state were handled incorrectly.

[1.1.0]

Added

- `Unity Netcode for Entities` package support. Animations and controllers can be synchronized using interpolated and predicted modes.
- New `Netcode Demo` sample with **Rukhanka** and `Netcode for Entities` collaboration showcase.
- Animator parameter aspect to simplify animator controller parameter data manipulation.

Changed

- Minimum `Entities` and `Entities.Graphics` packages version is 1.0.10.

Fixed

- State machine transitions with exit time 0 were handled incorrectly.
- Transitions with exit time 1 are looped contrary to Unity documentation. **Rukhanka** behavior changed to match `Mecanim` in this aspect.
- Various deprecated API usage warnings.

[1.0.3]

Added

- Adding authoring Unity.Animator and all used Unity.Animation in the baker dependency list.
- Extended animator controller logging with RUKHANKA_DEBUG_INFO which displays all states parameters and transitions of baked state machines.

Fixed

- Incorrect handling of very small transition exit time during state loops.

- Preventing NANs (division by zero) when transition duration is zero.
- Memory allocation error in PerfectHash tests.
- Controller parameters order in authoring animator does not coincide with generated `AnimatorControllerParameter` buffer.
- Empty animations to process buffer were handled incorrectly.
- Exit states of state machines are now handled properly.
- Animator state `Cycle Offset` treated as animation normalized time offset as in `Unity.Animator`.

[1.0.2]

Fixed

- Entities 1.0.0-pre.65 and Entities.Graphics 1.0.0-pre.65 support.

[1.0.1]

Added

- IEnableableComponent interface for AnimatorControllerLayerComponent and RigDefinitionComponent.
- Decsription of main **Rukhanka** entity components.

Changed

- `Crowd` and `Stress Test` samples now have control for skeleton visualization enabling (with RUKHANKA_DEBUG_INFO defined).
- `Crowd` and `Stress Test` samples now show total number of animated bones in scene.

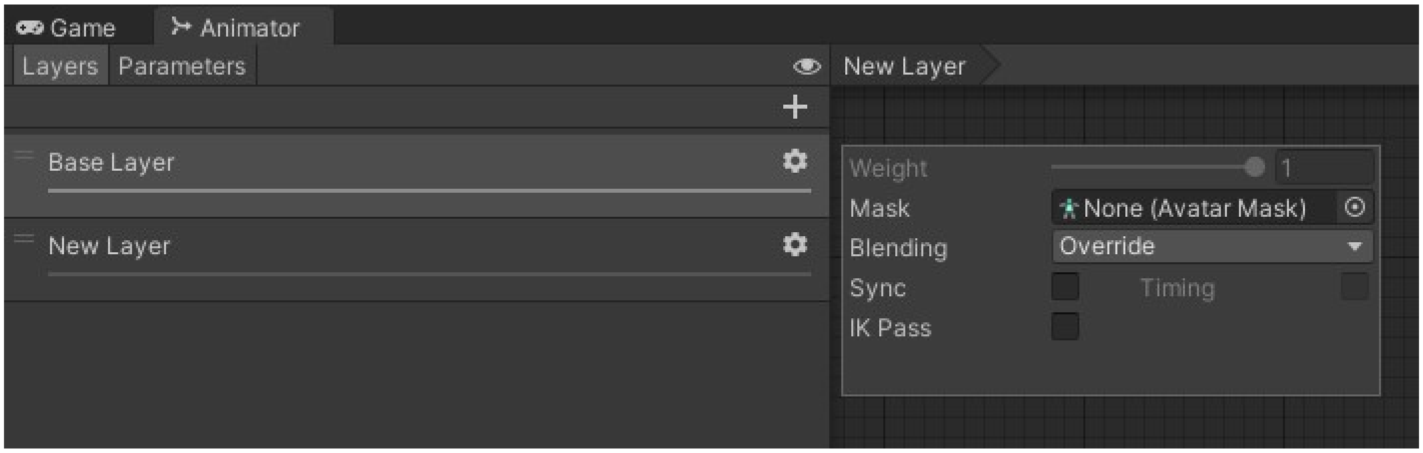
Fixed

- Incorrect handling handling of uniform scale in animations.

[1.0.0] - Initial release

Feature Support Tables

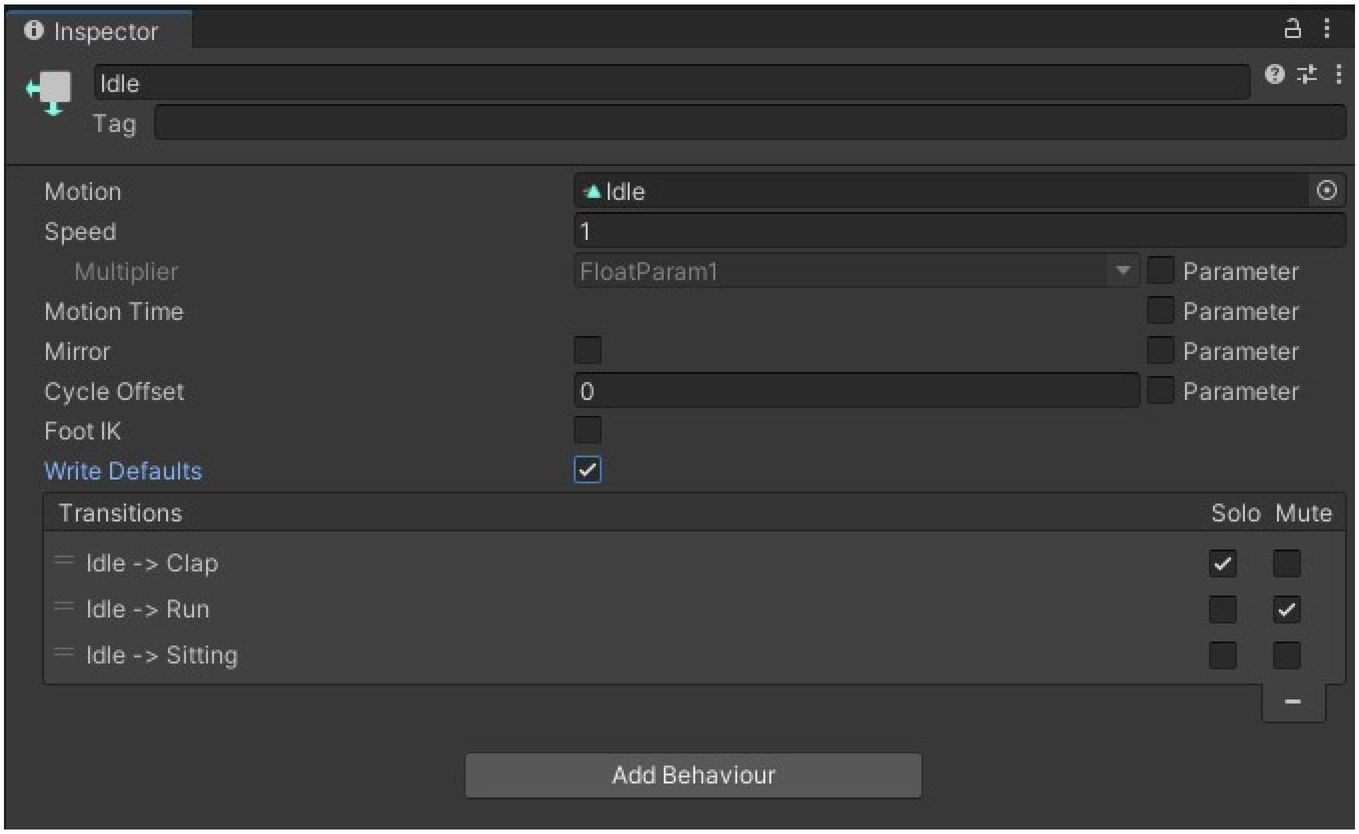
Animator Controller Layer



Weight 1
Mask None (Avatar Mask)
Blending Override
Sync
Timing
IK Pass

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

Animator State



Inspector

Idle

Tag

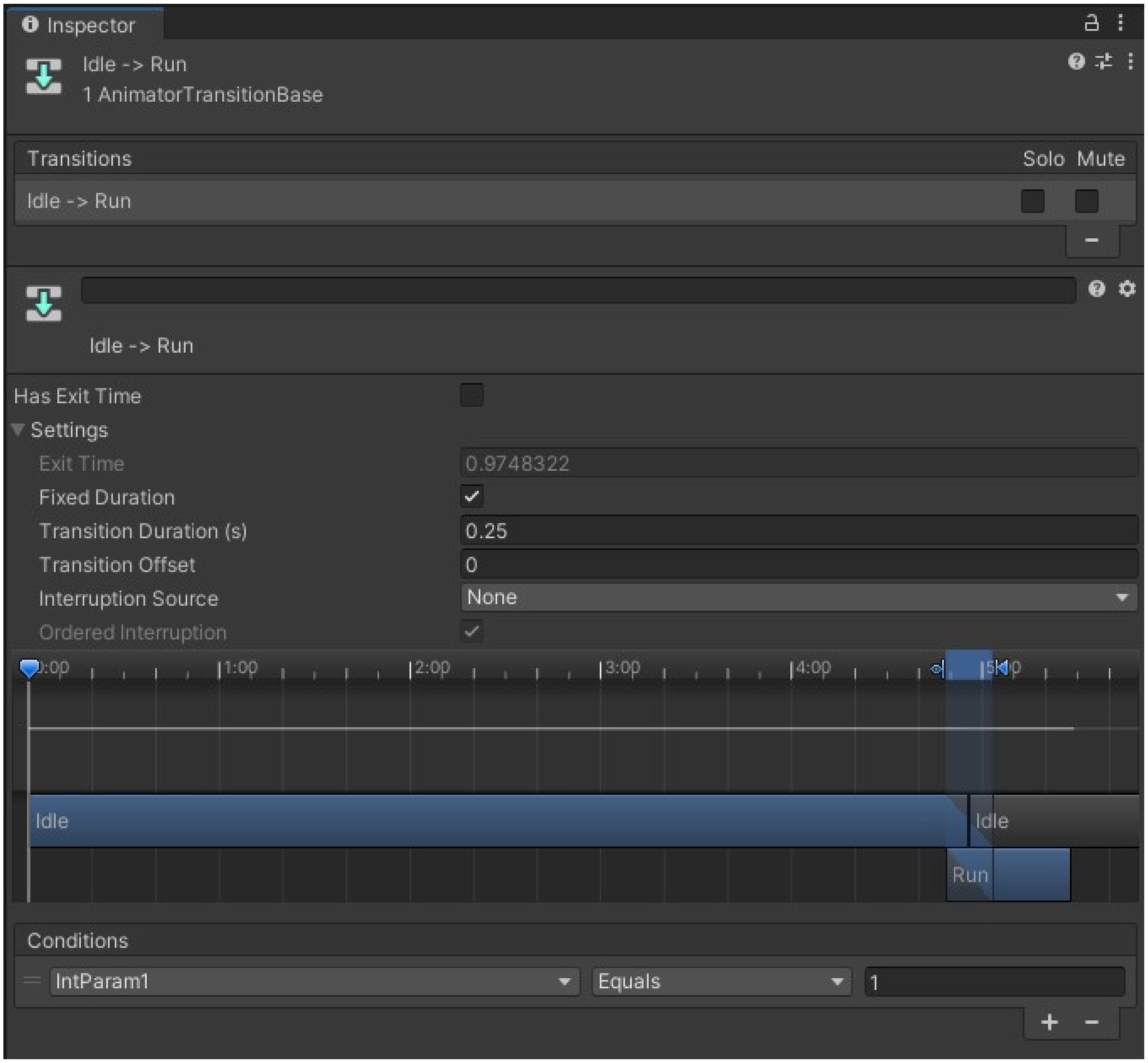
Motion	Idle	
Speed	1	
Multiplier	FloatParam1	☐ Parameter
Motion Time		☐ Parameter
Mirror	☐	☐ Parameter
Cycle Offset	0	☐ Parameter
Foot IK	☐	
Write Defaults	✓	

Transitions	Solo	Mute
Idle -> Clap	✓	☐
Idle -> Run	☐	✓
Idle -> Sitting	☐	☐

Add Behaviour

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

Animator Transition


Inspector

Idle -> Run
1 AnimatorTransitionBase

Transitions | Solo | Mute
Idle -> Run

Idle -> Run

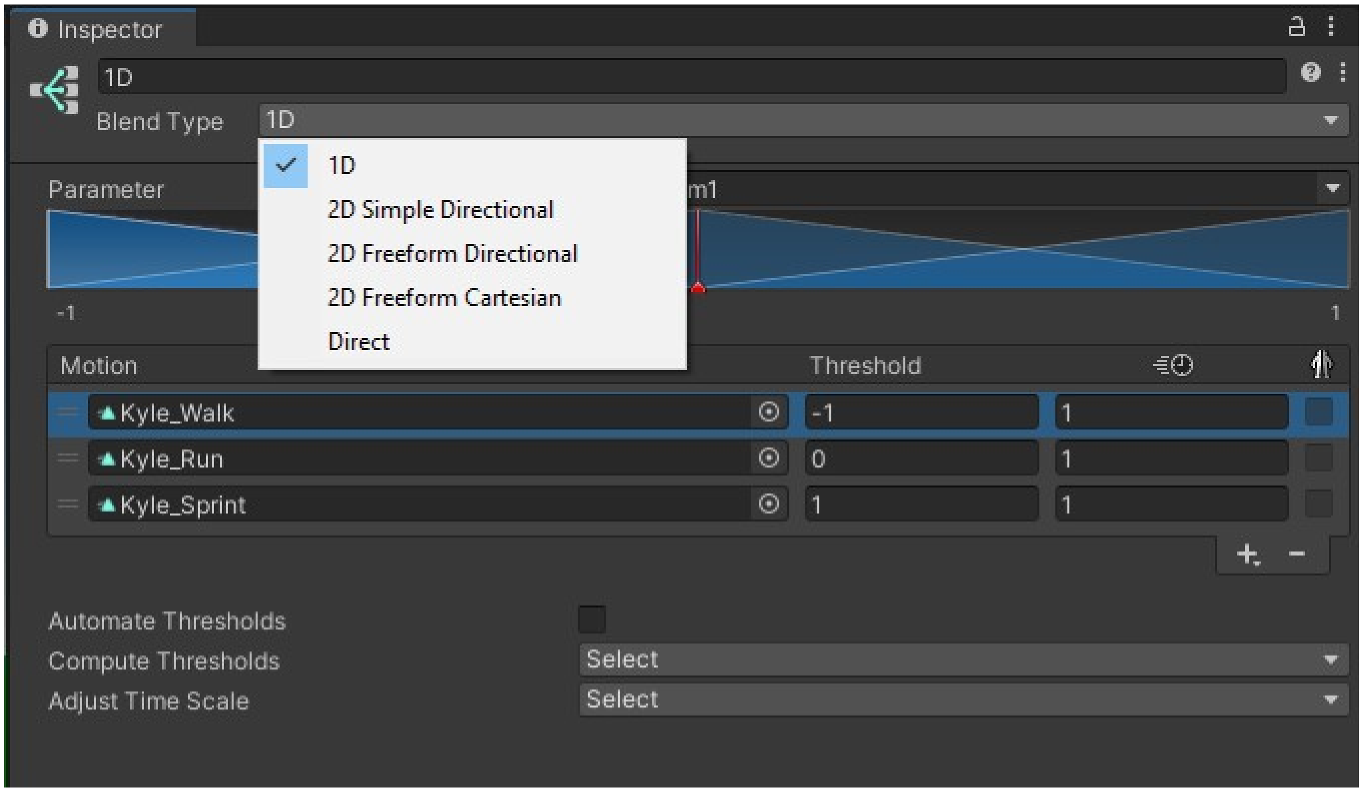
Has Exit Time
Settings
Exit Time 0.9748322
Fixed Duration ✓
Transition Duration (s) 0.25
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

Conditions
IntParam1 Equals 1

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



1D

Blend Type 1D

Motion	Threshold		
Kyle_Walk	-1	1	
Kyle_Run	0	1	
Kyle_Sprint	1	1	

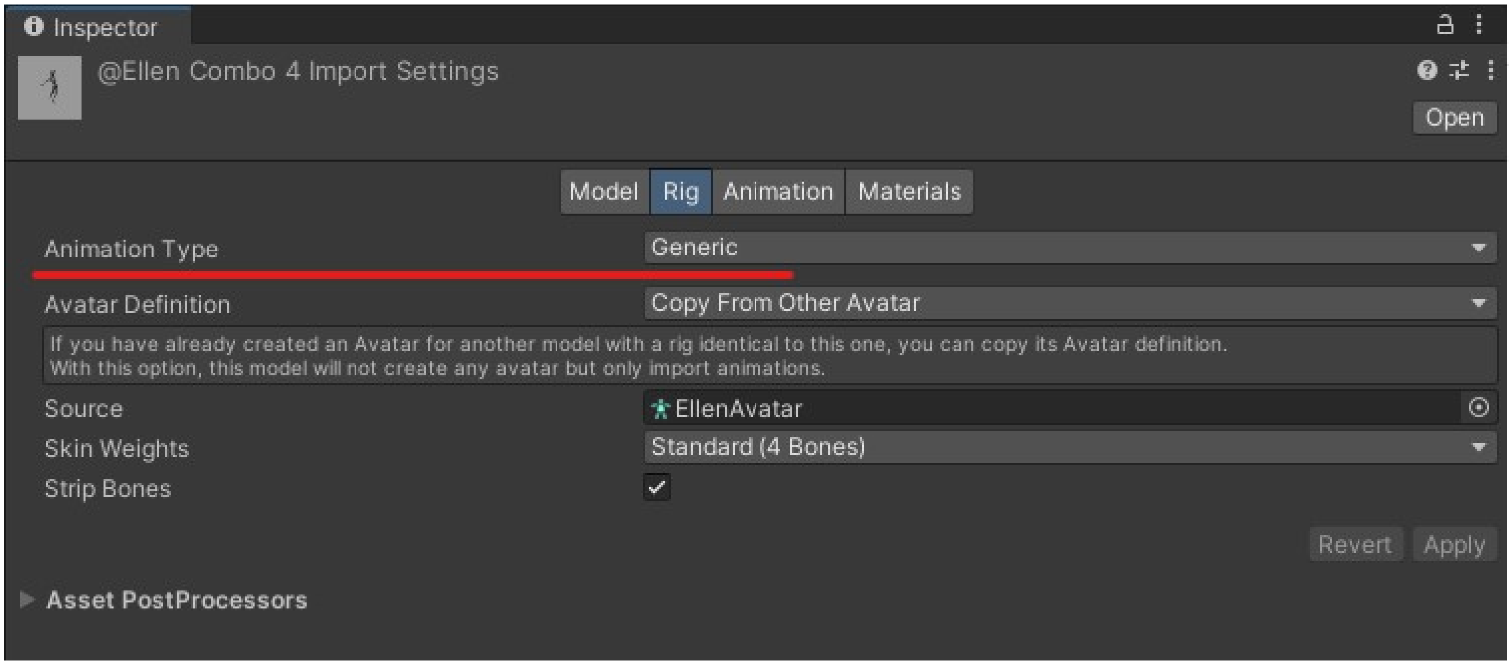
Automate Thresholds

Compute Thresholds Select

Adjust Time Scale Select

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

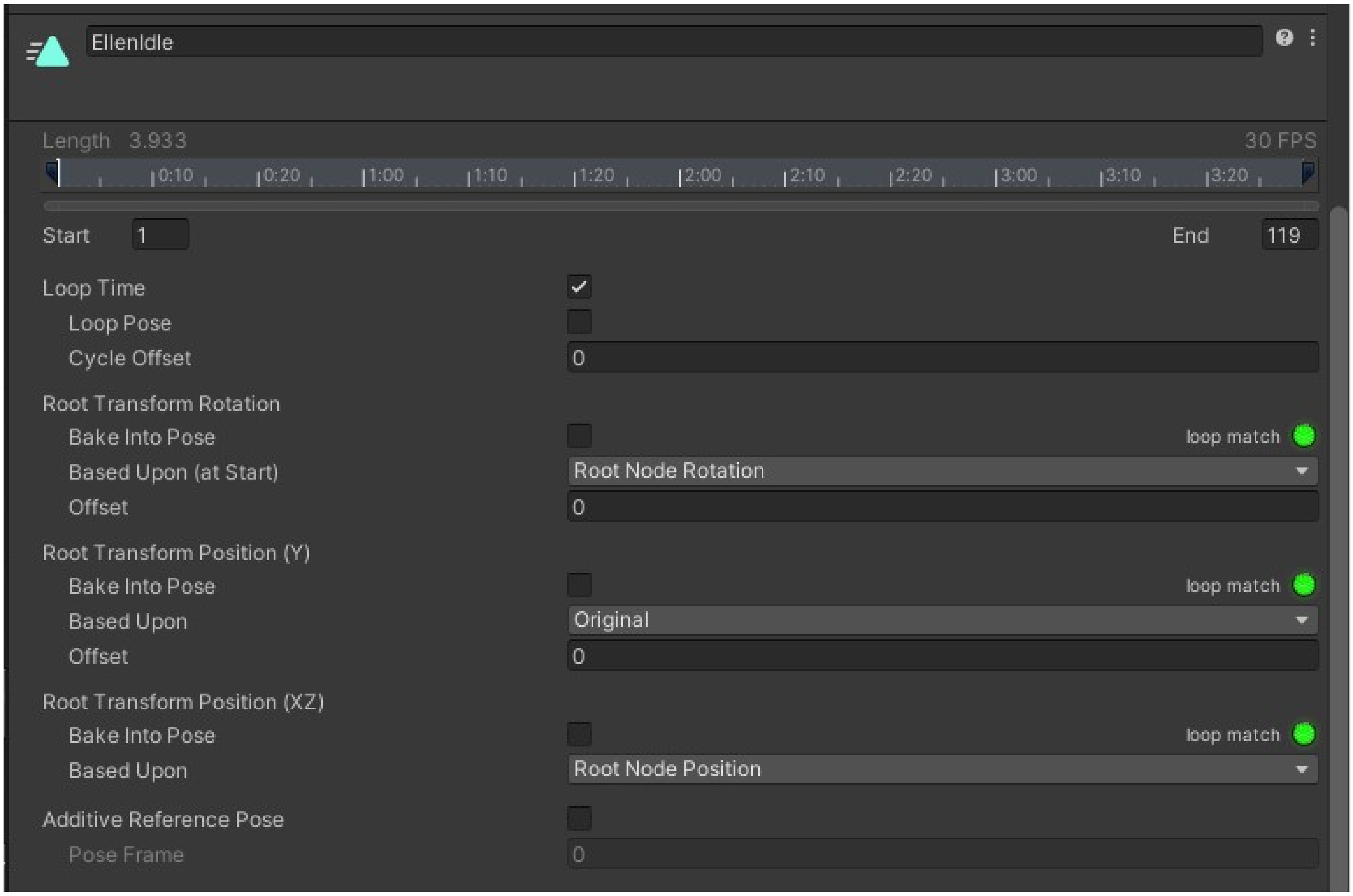
Animation Rig Properties


Inspector
@Ellen Combo 4 Import Settings

Animation Type: Generic
Avatar Definition: Copy From Other Avatar
If you have already created an Avatar for another model with a rig identical to this one, you can copy its Avatar definition.
With this option, this model will not create any avatar but only import animations.
Source: EllenAvatar
Skin Weights: Standard (4 Bones)
Strip Bones: ✓

Feature Name	Support Status	Additional Notes
Animation Type *Generic*	✅	
Animation Type *Legacy*	❌	
Animation Type *Humanoid*	✅	

Animation Properties


EllenIdle

Length 3.933 30 FPS

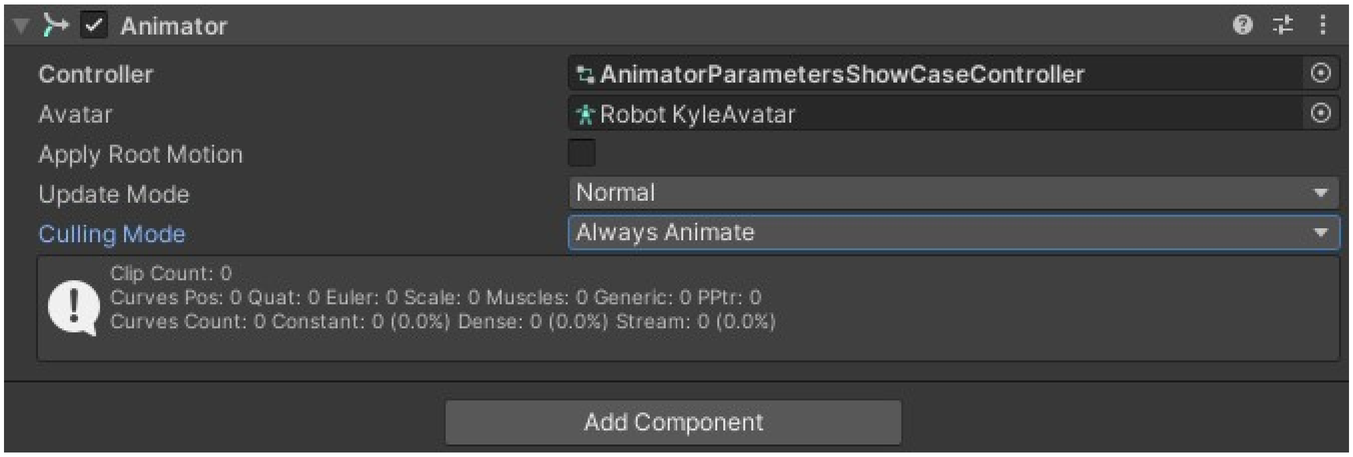
Start 1 End 119

Loop Time ✓
Loop Pose
Cycle Offset 0
Root Transform Rotation
Bake Into Pose loop match ●
Based Upon (at Start) Root Node Rotation
Offset 0
Root Transform Position (Y)
Bake Into Pose loop match ●
Based Upon Original
Offset 0
Root Transform Position (XZ)
Bake Into Pose loop match ●
Based Upon Root Node Position
Additive Reference Pose
Pose Frame 0

Feature Name	Support Status	Additional Notes
Loop Time	✅	
Loop Pose	✅	
Cycle Offset	✅	
Root Transform Rotation	✅	With all suboptions
Root Transform Position (Y)	✅	With all suboptions
Root Transform Position (XZ)	✅	With all suboptions
Additive Reference Pose	✅	
Pose Frame	✅	

Feature Name	Support Status	Additional Notes
Curves	✅	Documentation
Events	❌	
Mask	❌	
Motion	✅	Documentation

Animator Features



Controller: AnimatorParametersShowCaseController
Avatar: Robot KyleAvatar
Apply Root Motion
Update Mode: Normal
Culling Mode: Always Animate

```
Clip Count: 0
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0 Generic: 0 PPtr: 0
Curves Count: 0 Constant: 0 (0.0%) Dense: 0 (0.0%) Stream: 0 (0.0%)
```

Feature Name	Support Status	Additional Notes
Controller	✅	
Avatar	✅	Preparation steps are needed
Apply Root Motion	✅	Documentation
Update Mode	❌	
Culling Mode	❌	