Rukhanka ECS Animation System

Table of contents:

    - Fixed
    - Added
    - Changed
- [1.6.4] - 17.05.2024
    - Fixed
    - Added
- [1.6.3] - 16.03.2024
    - Changed
- [1.6.2] - 14.03.2024
    - Fixed
- [1.6.1] - 9.03.2024
    - Fixed
- [1.6.0] - 7.03.2024
    - Fixed
    - Added
    - Changed
- [1.5.1] - 10.02.2024
    - Fixed
    - Added
    - Changed
- [1.5.0] - 01.02.2024
    - Fixed
    - Added
    - Changed
- [1.4.2] - 15.12.2023
    - Fixed
    - Added
    - Changed
- [1.4.1] - 06.10.2023
    - Changed
    - Fixed
- [1.4.0] - 28.09.2023
    - Fixed
    - Added
    - Changed
- [1.3.1] - 11.08.2023
    - Fixed

v2.7.2

About Rukhanka

Rukhanka is an animation system based on the Entity Component System (ECS) for Unity's Data-Oriented Technology Stack (DOTS). It depends on the `Unity Entities` and the `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. **Rukhanka** has two animation computation engines: CPU (based on high performance `Burst` compiled multithreaded code) and GPU (based on compute shaders).

'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.com/invite/utdMamGbR8
- Support e-mail: support@rukhanka.com

Upgrading Rukhanka

If **Rukhanka** was installed in custom location (not via package manager) delete previous contents of the package before new version installation.

2.5.x → 2.6.0

- New skinned mesh baking approach makes several components obsolete. The following list describes the component changes in the **Rukhanka** source code:
  - `AnimatedRendererComponent`. This component was used on skinned mesh render entities to link them to the corresponding skinned mesh component and the animator. Render entities no longer exist, so this component no longer serves any purpose and has been removed.
  - `SkinnedMeshRenderEntity` dynamic buffer component. This buffer previously held references to all render entities related to a given skinned mesh entity. Since render entities are no longer used, this buffer also no longer serves any purpose and has been removed.
  - `AnimatedSkinnedMeshComponent`. This component, which held baked skinned mesh data, has been renamed to `SkinnedMeshRendererComponent` to better reflect its purpose. An entity with `SkinnedMeshRendererComponent` now holds all rendering-related data.

This change greatly simplifies the dynamic skinned mesh attachment workflow (there is no longer a need to manage the `SkinnedMesh → Root Bone → Render Entities` hierarchy), and all relevant data is now located on a single entity. The approach to applying `MaterialOverrides` is now the same as for ordinary mesh renderers.

2.4.x → 2.5.0

- *Deprecation of `IAspect`*. The `Entities` package marked the use of `IAspect` as deprecated starting from v1.4.0-exp.2. The usage of `IAspect` was removed from the codebase in `Rukhanka.Animation` v2.5.0. The migration process is straightforward:

Animator Parameters Aspect

Before v2.5.0:

```
struct MyAnimationProcessingJob: IJobEntity
{
    public FastAnimatorParameter moveSpeedParam;

    void Execute(AnimatorParametersAspect apa)
    {
        apa.SetFloatParameter(moveSpeedParam, 1.0f);
    }
}
```

After v2.5.0:

```
struct MyAnimationProcessingJob: IJobEntity
{
    public FastAnimatorParameter moveSpeedParam;

    void Execute(DynamicBuffer<AnimatorControllerParameterComponent>
animatorParametersBuf,
                 AnimatorControllerParameterIndexTableComponent
animatorParametersIndexTable)
    {
        var apa = new AnimatorParametersAspect(animatorParametersBuf,
animatorParametersIndexTable);
        apa.SetFloatParameter(moveSpeedParam, 1.0f);
    }
}
```

Animator State Query Aspect

Before v2.5.0:

```
struct MyAnimatorStateQueryJob: IJobEntity
{
    void Execute(AnimatorStateQueryAspect animatorStateQuery)
    {
        //  Use animatorStateQuery object
    }
}
```

After v2.5.0:

```
struct MyAnimatorStateQueryJob: IJobEntity
{
    void Execute(DynamicBuffer<AnimatorControllerLayerComponent> layers)
    {
        var animatorStateQuery = new AnimatorStateQueryAspect(layers);
        //  Use animatorStateQuery object
    }
}
```

The same approach should be used for idiomatic `foreach` migration.

2.1.x → 2.2.0

- `Entities.Graphics` deformation shader is deprecated to use with **Rukhanka Animation System**. A console warning will be issued if such shader is detected on skinned mesh shader material. Please upgrade your deformation shaders as described in the documentation. Note that new deformation features like in-place skinning, half-precision deformation data, and dual quaternion skinning will not work with `Entities.Graphics` deformation shader.

1.9.x → 2.0.0

- Due to the massive amount of internal changes all animated entities and blob assets require rebake.

1.6.x → 1.7.0

- Make sure that new `RigDefinitionAuthoring` `Rig Config Source` field is set to `From Animator` (default behavior). New `User Defined` mode should be used to work without Unity's `Animator Controller`.

1.5.1 → 1.6.0

- With introducing of animation culling ability, previously configured authoring prefabs can behave incorrectly. Please, carefully read Animation Frustum Culling section of documentation and make necessary prefab adjustments.

- With introducing of skinned mesh renderer bounding box recalculation ability, previously configured authoring prefabs can behave incorrectly. Please, carefully read Renderer Bounding Box Recalculation section of documentation and make necessary prefab adjustments.

1.5.0 → 1.5.1

- `RebuildOutdatedBonePoses` function was removed from the `public` accessibility level. `AnimationStream` will rebuild outdated bone poses automatically on `Get` calls. It is now derived from an `IDisposable` interface, and disposal is required after its usage.

1.4.0 → 1.5.0

- The internal package name has been changed to `com.rukhanka.animation`. You need to adjust your assembly references if **Rukhanka** is referenced by name.
- Entity bone stripping mode needs to be configured again due to the `RigDefinitionAuthoring` UI change.

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 1.4.3+ (installed automatically as dependency)
- Unity `Entities.Graphics` package version 1.4.16+ (installed automatically as dependency)
- HDRP or URP as required by `Entities.Graphics` package
- [Optional] `Netcode for Entities` package for network animation synchronization support

> **TIP**
>
> Use the latest `Entities` and `Entities.Graphics` packages available. They often contain important fixes and improvements.

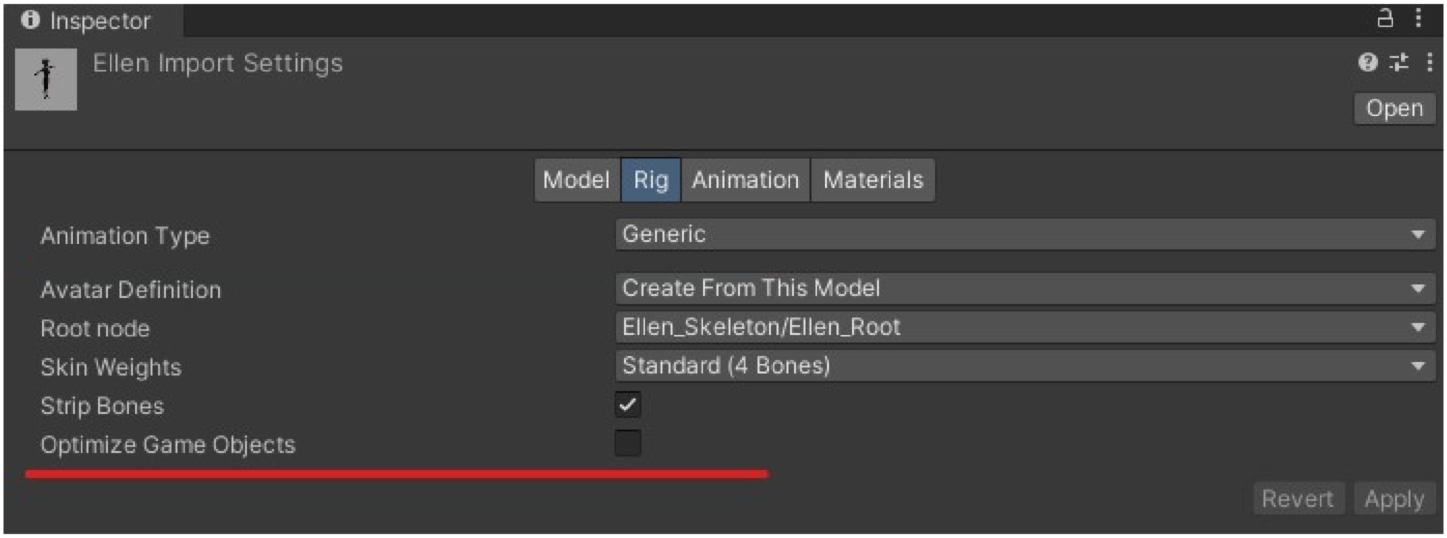
Animated Object Setup

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

Model Importer

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

- Uncheck `Optimize Game Objects` checkbox. **Rukhanka** need all bone game objects in baking phase.

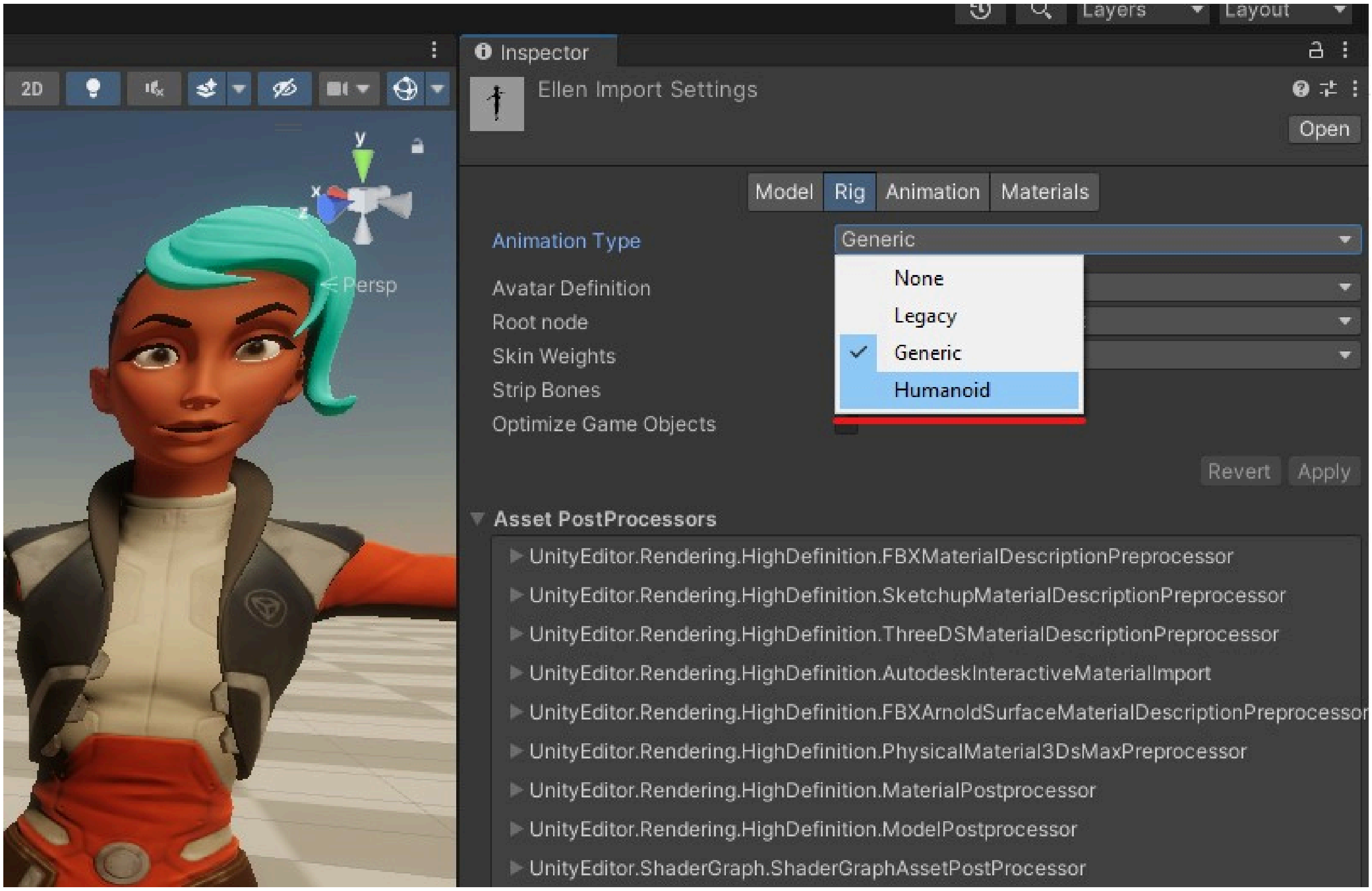


Rig Definition

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**.

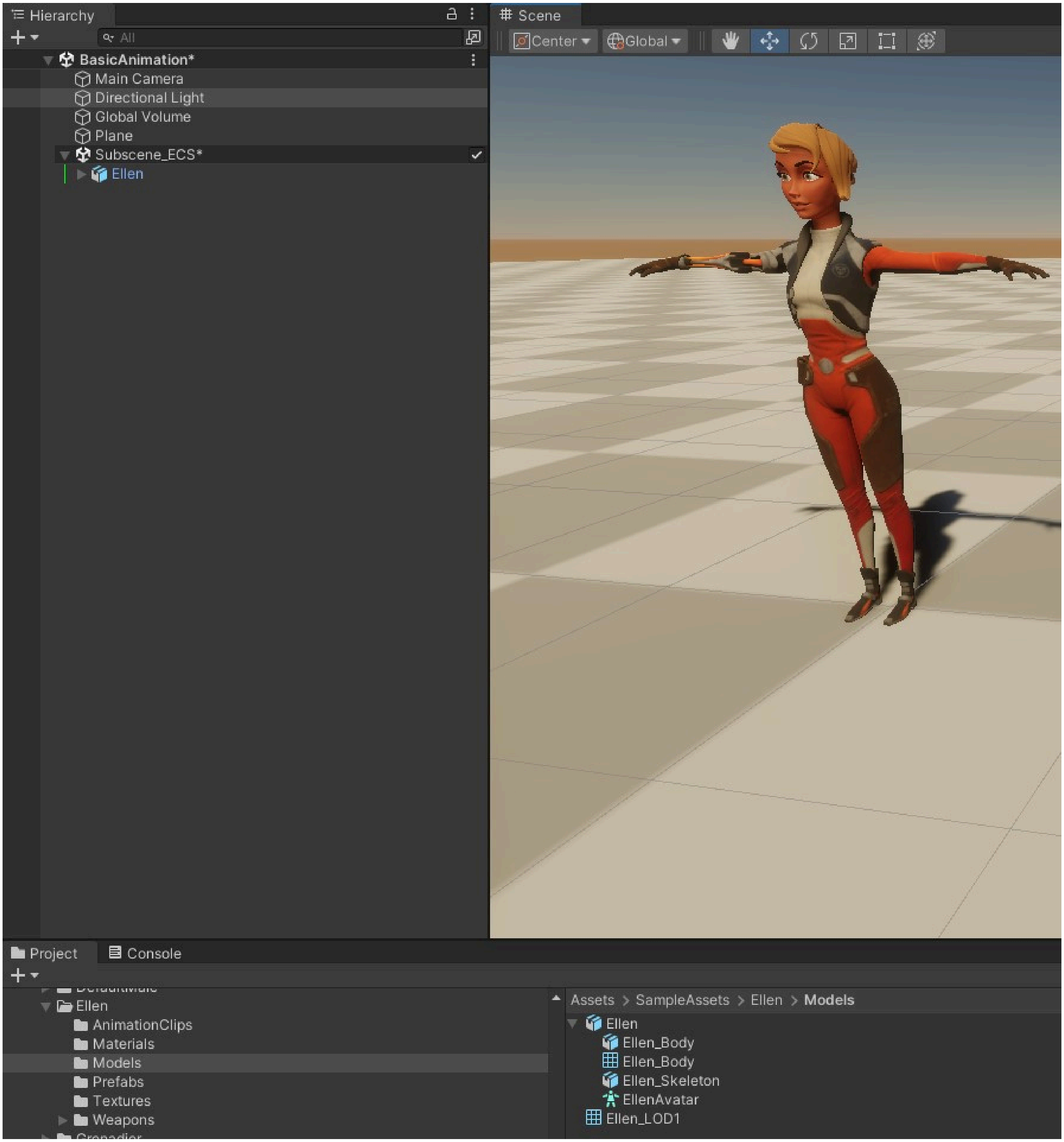


2. For `Humanoid` models configuration process is ended already. For `Generic` models you need to switch `Animation Type` of model back to `Generic` and press `Apply`.

Authoring Object Setup

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

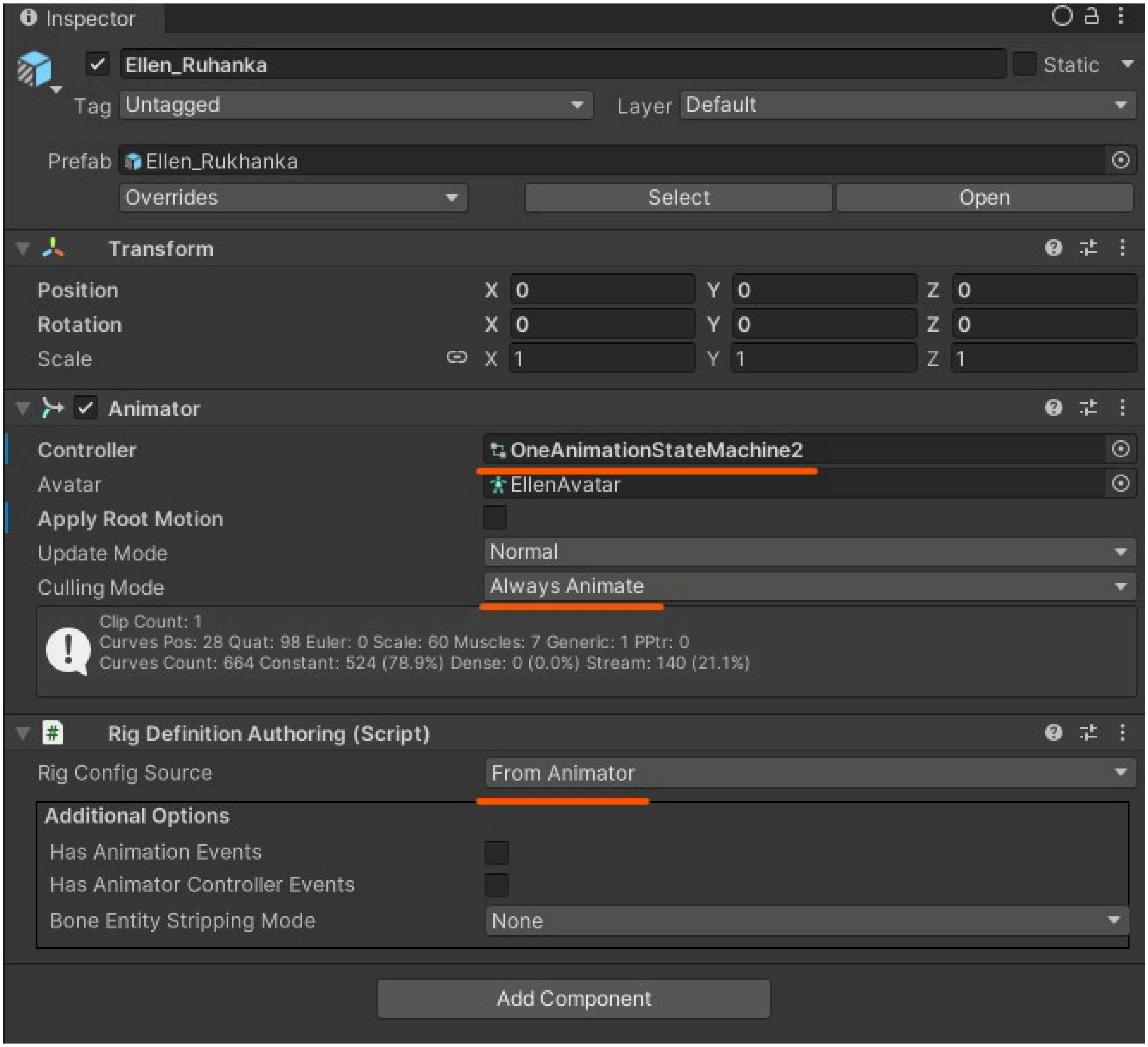
1. Place your animated object inside `Entities` Subscene. For detailed description of this step refer [Entites Package documentation](#).



2. Add `Rig Definition Authoring` component to newly created object. Make sure that `Rig Config Source` is set to `From Animator`.

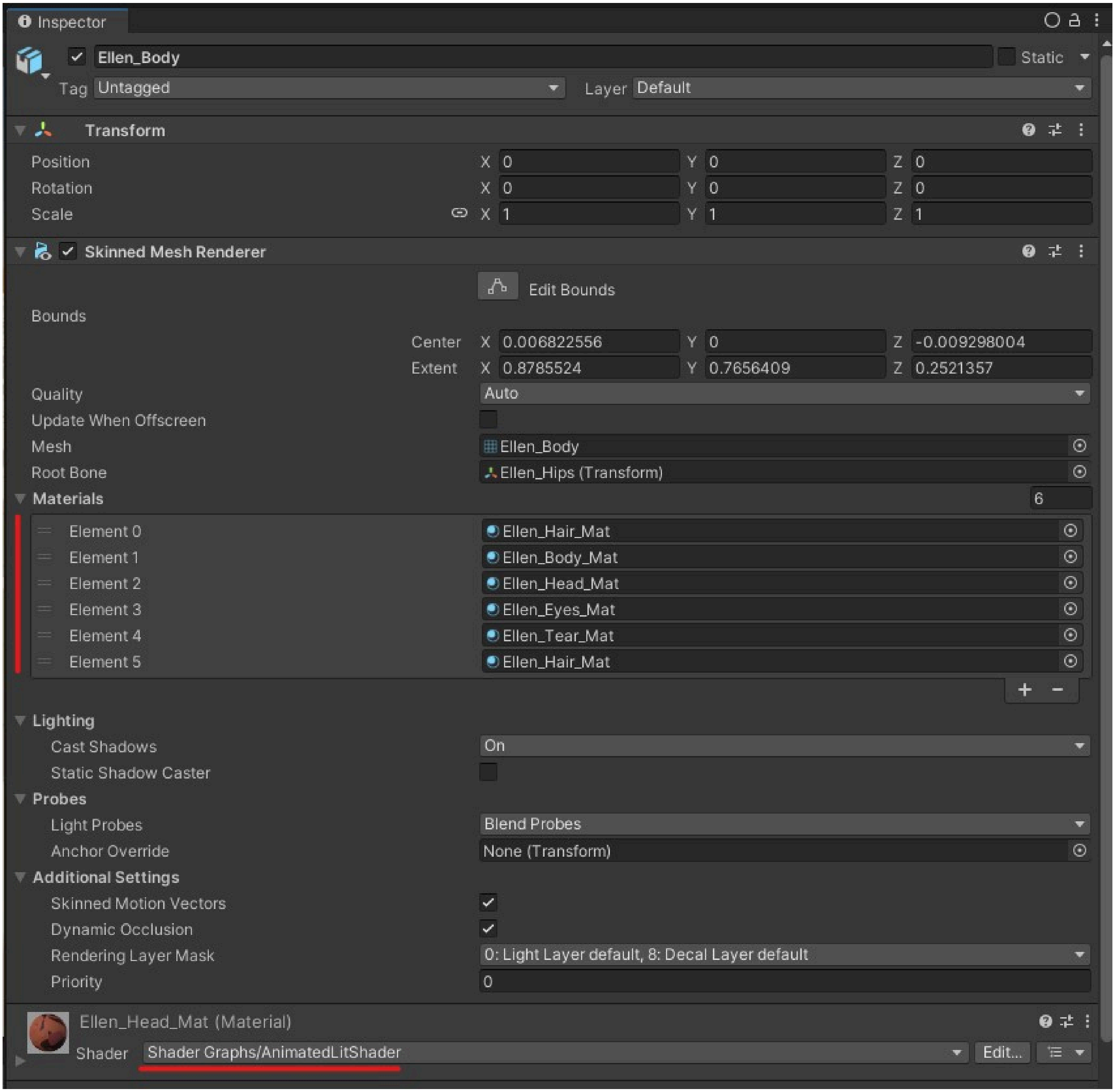
3. Create standard `Animator Controller` and fill it as you wish (one state with one animation will be a good start).

4. Switch culling mode to `Always Animate`.



Shaders And Materials

Rukhanka does not render animated objects. It only prepares skin matrices for skinned meshes that are processed by `Rukhanka Deformation System`. To be able to render deformed meshes correctly it is required to make a special deformation-aware shader. 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.



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 required animations.

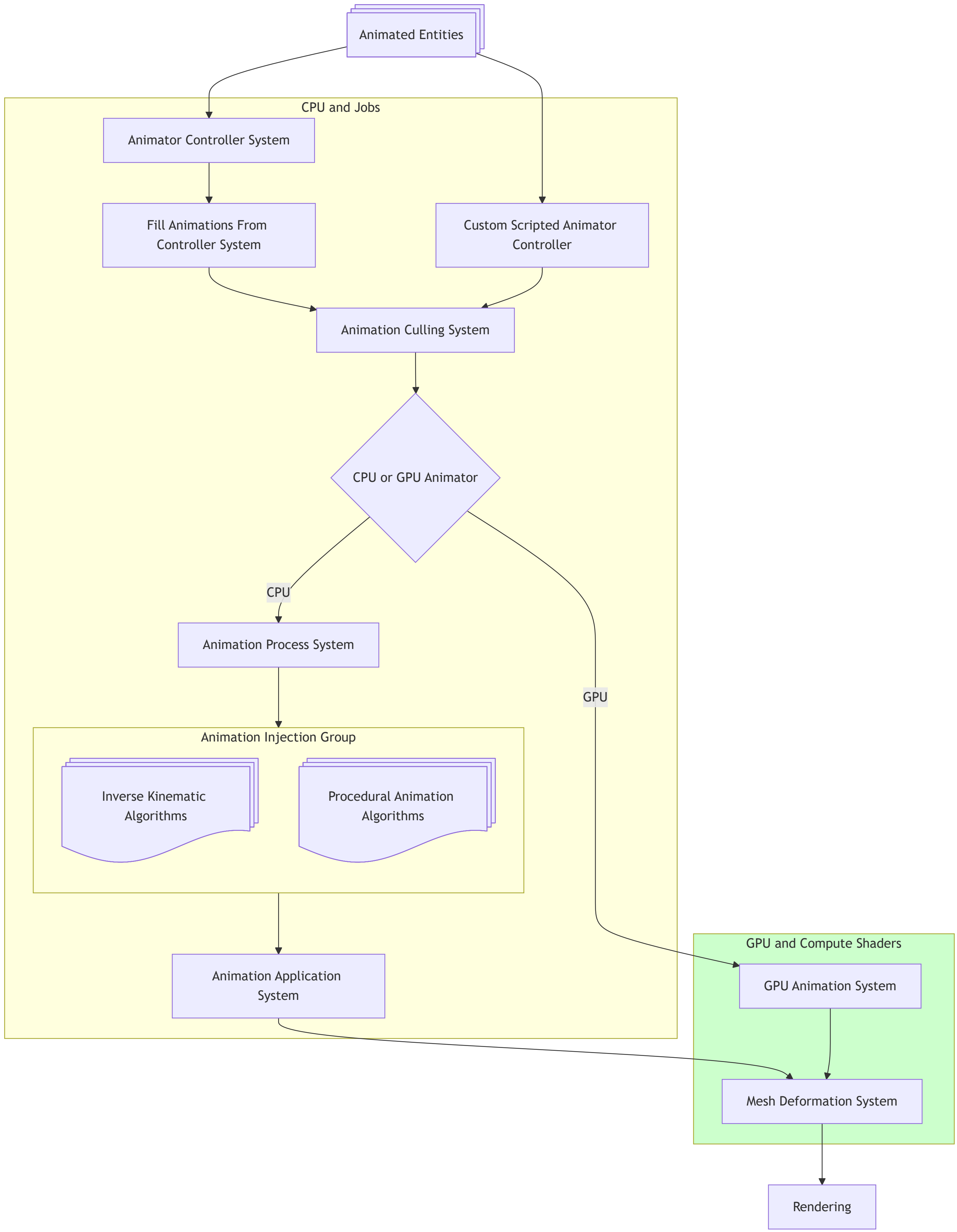
IMPORTANT: There is not 100% Unity's `Mecanim` feature support. Please consult Feature Support Tables for complete information.

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

Rukhanka Runtime Architecture

Overview

A high-level overview of the runtime **Rukhanka** architecture looks as follows:



Animated Entities
CPU and Jobs
Animator Controller System
Fill Animations From Controller System
Custom Scripted Animator Controller
Animation Culling System
CPU or GPU Animator
CPU
GPU
Animation Process System
Animation Injection Group
Inverse Kinematic Algorithms
Procedural Animation Algorithms
Animation Application System
GPU and Compute Shaders
GPU Animation System
Mesh Deformation System
Rendering

Controller

Animator Controller System

This system works with `AnimatorControllerLayerComponent` and advances controllers baked from Unity's `Animator Controllers`.

Fill Animations From Controller System

This system uses the output produced by the `Animator Controller System` to fill the `AnimationToProcessComponent` buffer with animations that need to be played in the current frame for a given entity.

> **INFO**
>
> The `AnimatorControllerSystem` and `FillAnimationsFromControllerSystem` are optional systems. They can (and should) be removed if your are using your own scripted animator.

Animation Culling System

If the entity bounding box outside configured frustums, skip animation processing for this entity.

CPU Animation Processing Path

> **INFO**
>
> CPU animation path is taken if the animated entity does not have `GPUAnimationEngineTag` or has it disabled.

Animation Process System

Animation tracks for requested animations are sampled and blended to define bone positions for a given animation frame.

Animation Injection Group

`RukhankaAnimationInjectionSystemGroup` is a predefined place for systems that want to alter bone poses before they will be applied to the corresponding bone entities and skinned meshes. All IK algorithms reside there.

Animation Application System

Animated bone poses are used to set the position/rotation/scale of corresponding bone entities. Also, skin poses are created from animated bone poses and they are written into the `SkinPose` buffer.

GPU Animation Processing Path

> **INFO**
>
> GPU animation path is taken if the animated entity has enabled the `GPUAnimationEngineTag` component.

GPU Animation System

This is a combined equivalent of the CPU's `Animation Process System` and `Animation Application System`. GPU path does not have a special animation injection modification point (and, as a result, IK) and produces and writes skin poses directly to the GPU buffer for later use by `MeshDeformationSystem`.

Mesh Deformation System

After animation data has been calculated (either by CPU or GPU paths) it is used to deform skinned meshes. This process is always performed on GPU using compute shaders.

> **INFO**
>
> Through the documentation CPU and GPU animator differences are presented in tabs:
>
> **CPU Animator** **GPU Animator**
>
> CPU Animator specific information

Animation Engines

The animation engine is a part of **Rukhanka** that reads, processes and blends animation tracks for the animated rigs.

Rukhanka has two animation engine implementations:

- One that is completely implemented on parallel jobs and `Burst`-compatible C# code. In the documentation, this engine is referred to as `CPU Animator`.
- Another with the core routines implemented using compute shaders. In the documentation, this engine is referred to as `GPU Animator`.

The reference **Rukhanka** animation engine is the `CPU Animator`. The `GPU Animator` is closely followed by the `CPU Animator` and produces exactly the same animation results, but due to the CPU-GPU interprocess communication aspects, cannot provide all functionality available from the `CPU Animator`.

Animators Feature Support Table

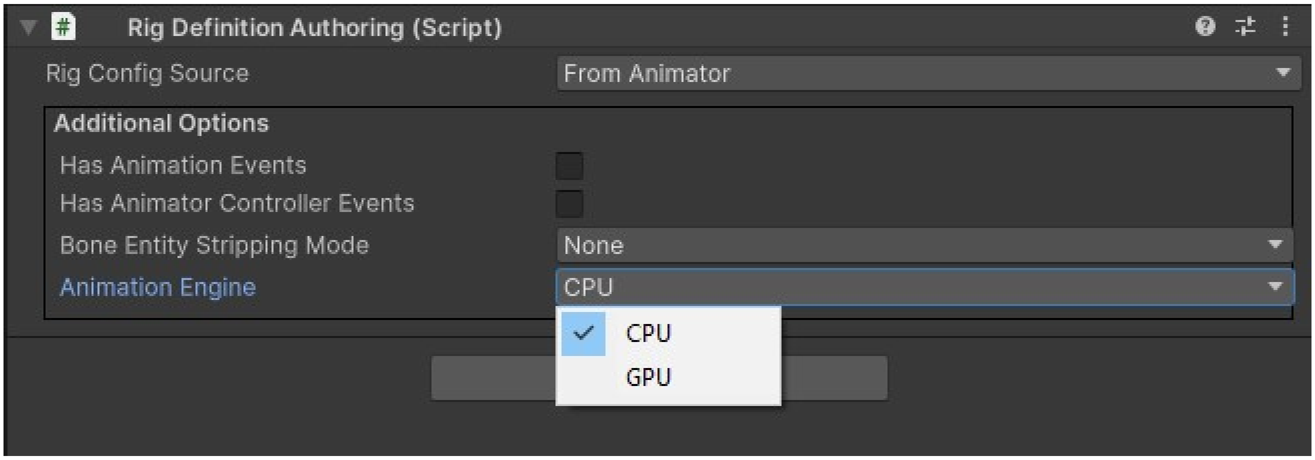
The following table shows available animator features and their support status.

Feature Name	`CPU Animator` Support Status	`GPU Animator` Support Status
Legacy animations	❌	❌
Generic animations and avatars	✅	✅
Humanoid animations and avatars	✅	✅
Weighted animation blending	✅	✅
Avatar mask application	✅	✅
Loop pose calculation	✅	✅
Animation Events	✅	✅

Feature Name	`CPU Animator` Support Status	`GPU Animator` Support Status
Altering animation results	✅	❌
Inverse Kinematics	✅	❌
Root motion	✅	❌
User animation curves	✅	❌
Blend shapes	✅	❌
Bone position writeback	✅	❌
Working with physics	✅	❌
Working with Netcode	✅	✅
Animation frustum culling	✅	✅
Render bounding box update	✅	❌
Bone attachements	✅	✅

Animation Engine Selection

`Rig Definition Authoring` has the `Animation Engine` dropdown, where you can choose animator for a baked entity:



Animation Engine Runtime Switching

Rukhanka can switch between CPU and GPU animators at runtime. The switch will be performed seamlessly and can be made at any time (even during animation playback). Use `GPUAnimationEngineTag` enableable component state to switch between animators:

- If the component is enabled, **Rukhanka** will use `GPUAnimator` for that entity.
- If the component is disabled, **Rukhanka** will use `CPUAnimator` for that entity.

Shaders with Deformations

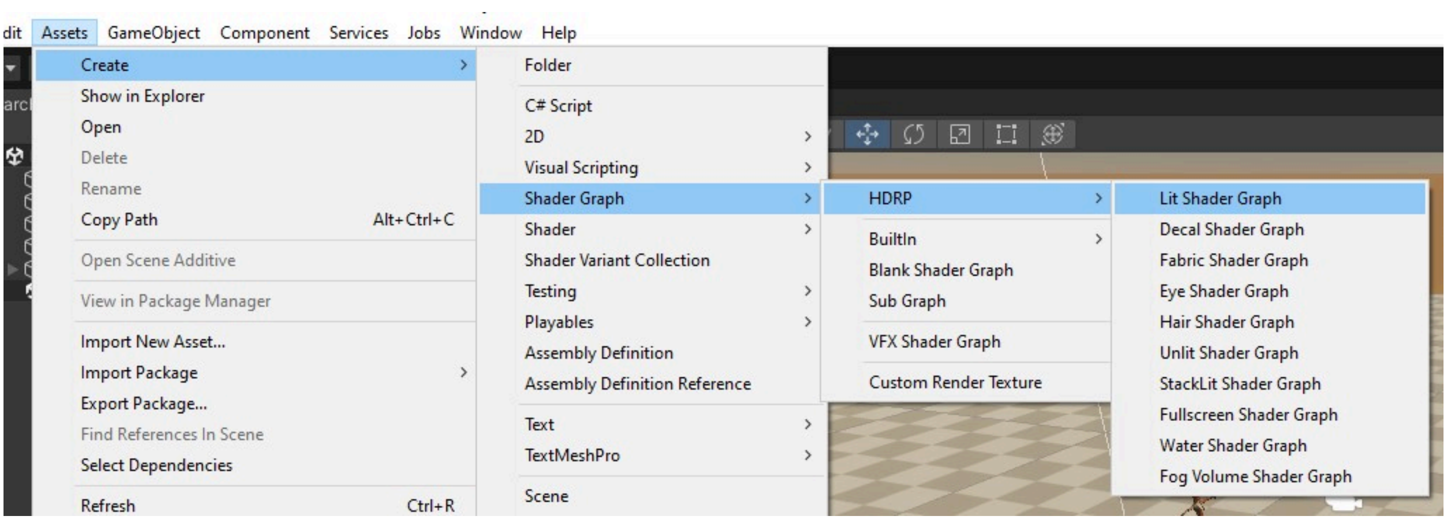
For the correct rendering of skinned meshes deformed by **Rukhanka**, a deformation-aware shader should be created. To make this task `Unity Shader Graph`, `Amplify Shader Editor`, or `Better Shaders` tool can be used.

> **TIP**
>
> Simple deformation-compatible shaders can be found in **Rukhanka** samples

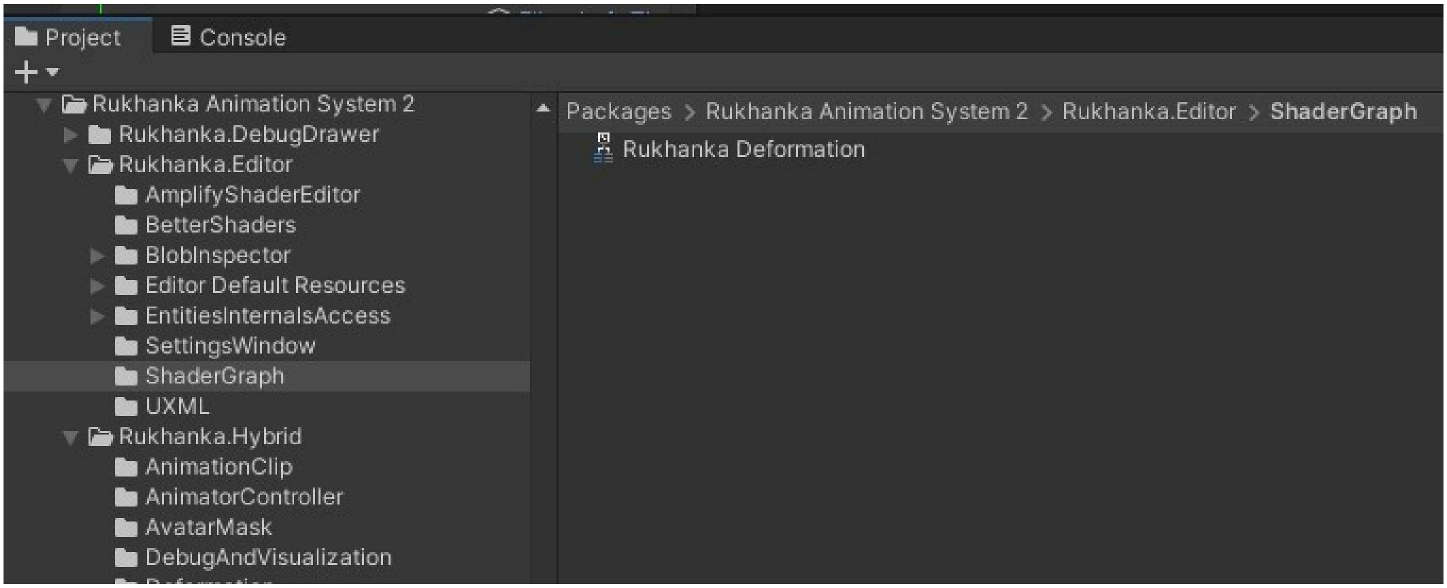
Unity Shader Graph

Creating deformation-compatible shaders using `Unity Shader Graph` is straightforward:

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

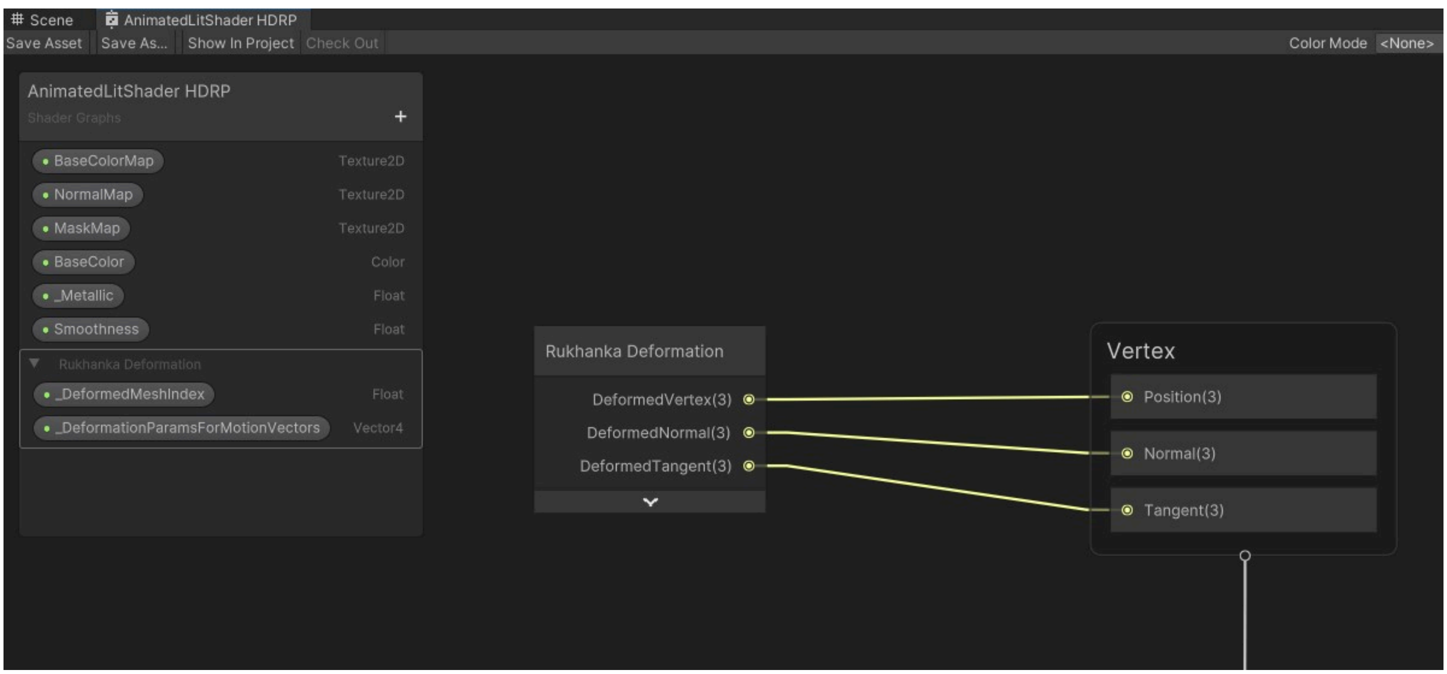


2. Navigate to the `Packages/Rukhanka Animation System 2/Rukhanka.Editor/ShaderGraph/` folder in the project inspector window.



3. Add `Rukhanka Deformation` subgraph to the created `ShaderGraph` by dragging and dropping it from the project inspector.

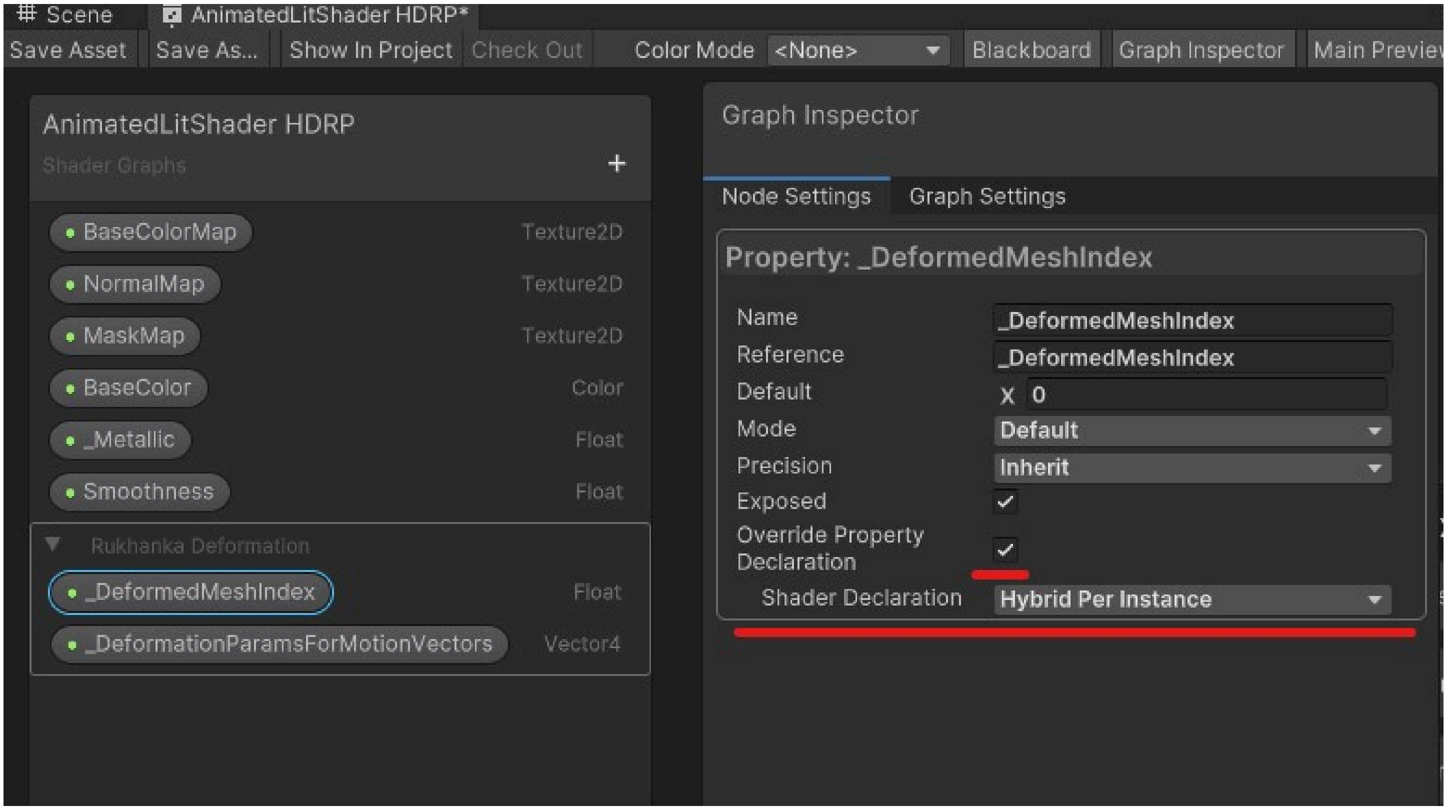
4. Connect position, normal, and tangent output ports of the `Rukhanka Deformation` node to the corresponding input ports of the master node.



5. Create two shader parameters:

- `_DeformedMeshIndex` with type `float`.
- `_DeformationParamsForMotionVectors` with type `Vector4`.

Make both parameters declared as `Hybrid Per Instance`.



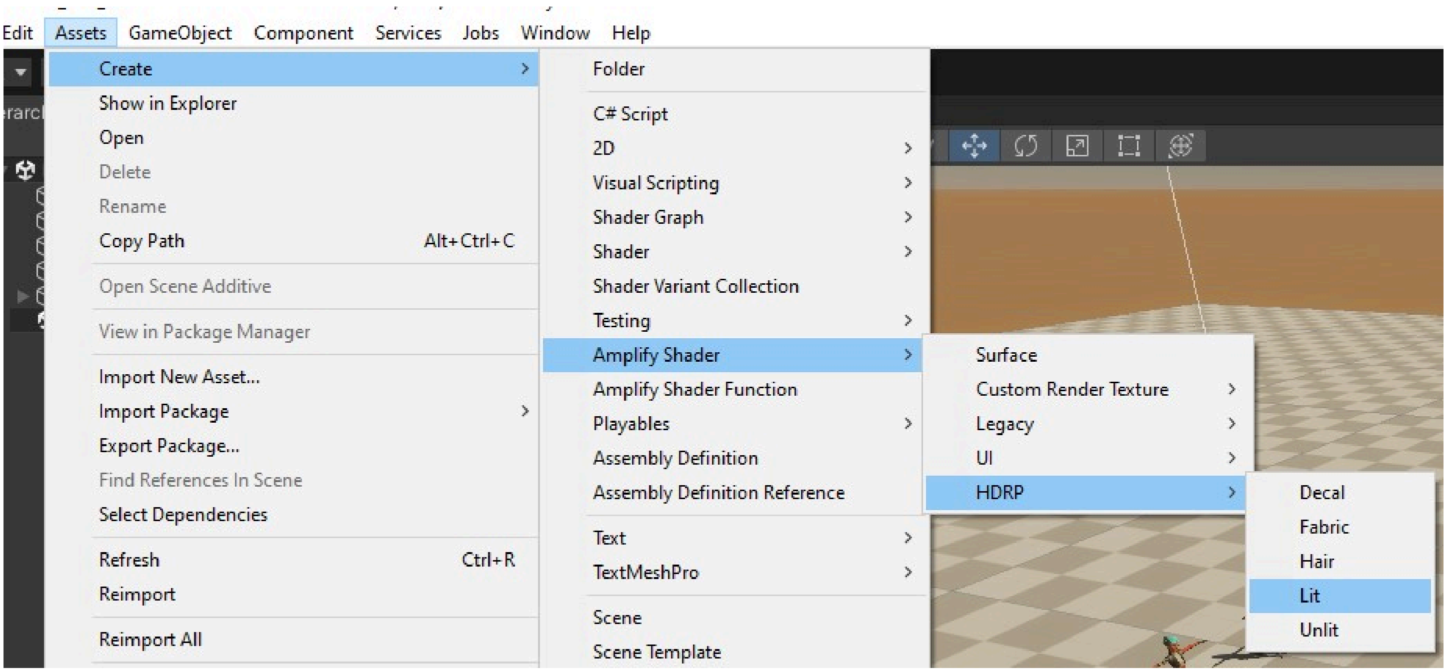
5. Save and assign this newly created shader to the materials of skinned mesh renderers.

> **TIP**
>
> Production Ready Shaders included in the `Shader Graph` sample assets can be used as animated shader templates

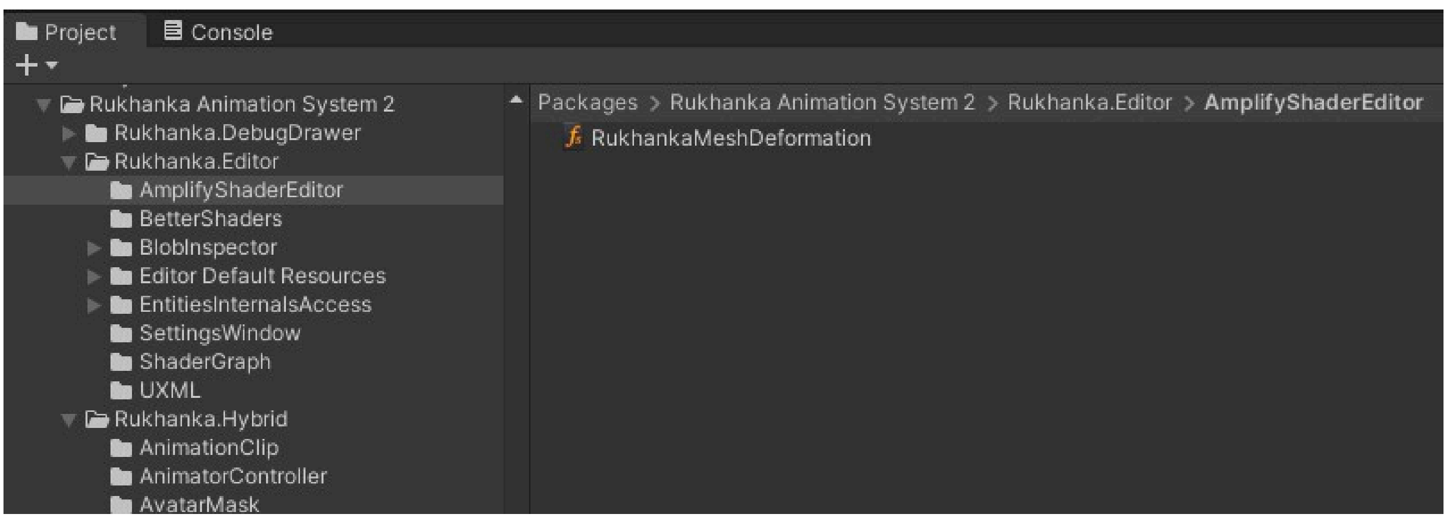
Amplify Shader Editor

The process of creating deformation aware shader in `Amplify Shader Editor` is also simple:

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



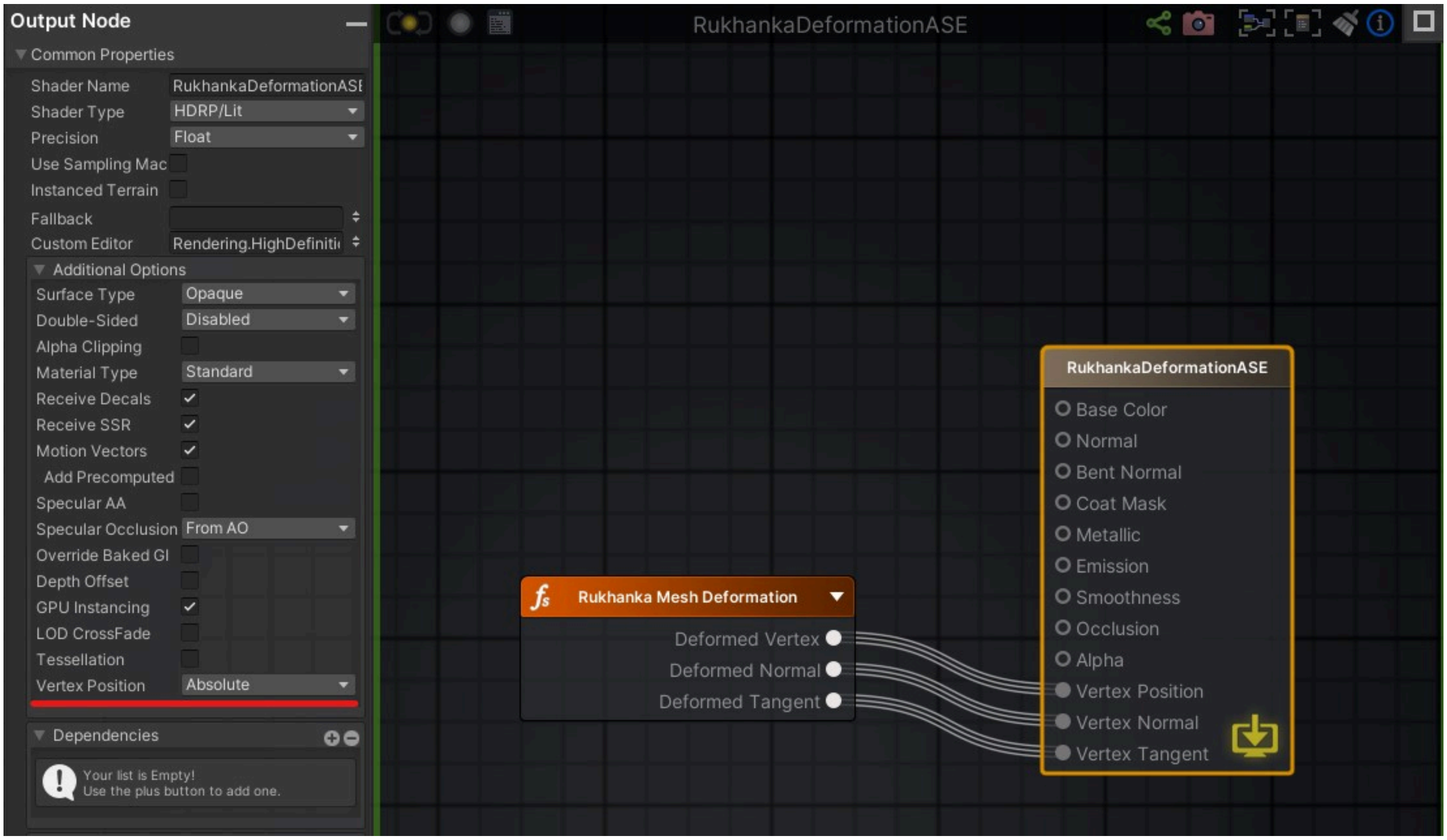
2. Navigate to the `Packages/Rukhanka Animation System 2/Rukhanka.Editor/AmplifyShaderEditor/` folder in the project inspector window.



3. Add `RukhankaMeshDeformation` amplify shader function to the created shader by dragging and dropping it from the project inspector.

4. Connect the vertex position, normal, and tangent output ports of the `Rukhanka Mesh Deformation` node to the corresponding input ports of the master node.

5. Set vertex position mode of the master node to `Absolute`.



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

Better Shaders

The `Better Shaders` is a source-based shader generation tool. **Rukhanka** provides the shader snippets that can be included in your `BetterShaders` shader.

Navigate to the **Rukhanka** installation directory and copy `Rukhanka.Editor/BetterShaders/StackableRukhankaDeformation.surfshader` and `Rukhanka.Editor/BetterShaders/StackableRukhankaDeformationMotionVectors.surfshader` files to the directory with your `BetterShaders` shader. Use `StackableRukhankaDeformation.surfshader` or `StackableRukhankaDeformationMotionVectors.surfshader` (depending on motion vector requirements) in the **first** line of `BEGIN_SUBSHADERS/END_SUBSHADERS` block. An example of a lit shader can be as follows:

```
BEGIN_OPTIONS
   Stackable "False"
   ShaderName "BetterShaders/LitAnimated"
END_OPTIONS

BEGIN_SUBSHADERS
   // Must be first line in subshader list
```

```
    "StackableRukhankaDeformation.surfshader"

    // Use this line instead if RUKHANKA_ENABLE_DEFORMATION_MOTION_VECTORS script symbol is defined
    //"StackableRukhankaDeformationMotionVectors.surfshader"

    "Lit.surfshader"
END_SUBSHADERS
```

Direct Source Code Modification

The shader source code can be modified to properly work with **Rukhanka deformations**. The process can be described as follows:

Shader Properties Section

Define the following two shader variables:

```
[HideInInspector]_DeformedMeshIndex("Deformed Mesh Buffer Index Offset", Float) = 0
[HideInInspector]_DeformationParamsForMotionVectors("Deformation Parameters", Float) = 0
```

For each pass

1. Enable the DOTS instancing variant

```
#pragma multi_compile _ DOTS_INSTANCING_ON
```

2. Add variables to the `UnityPerMaterial` constant buffer

Find or create the `UnityPerMaterial` constant buffer declaration and add both variables:

```
CBUFFER_START(UnityPerMaterial)
    float  _DeformedMeshIndex;
    float4 _DeformationParamsForMotionVectors;
CBUFFER_END
```

3. Add variables to the `UserPropertyMetadata` DOTS instancing section

Find or create the `UserPropertyMetadata` DOTS instancing props section and add both variables:

```
#ifdef UNITY_DOTS_INSTANCING_ENABLED
UNITY_DOTS_INSTANCING_START(UserPropertyMetadata)
  UNITY_DOTS_INSTANCED_PROP(float, _DeformedMeshIndex)
  UNITY_DOTS_INSTANCED_PROP(float4, _DeformationParamsForMotionVectors)
UNITY_DOTS_INSTANCING_END(UserPropertyMetadata)
#endif
```

4. Include the deformation compute file after the `UserPropertyMetadata` section:

```
#include
"Packages/com.rukhanka.animation/Rukhanka.Runtime/Deformation/Resources/ComputeDeformedVertex
```

5. Modify vertex shader input structure. Add a `vertexID` field with the `SV_VertexID` semantic:

```
struct Attributes
{
  float4 vertex         : POSITION;
  float3 normal         : NORMAL;
  float4 tangent        : TANGENT;
  float4 texcoord0      : TEXCOORD0;
  uint vertexID         : SV_VertexID;
  UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

6. Modify vertex shader entry function. At the beginning of the vertex shader entry function, add a call to `ComputeDeformedVertex_float`:

```
Varyings LitPassVertex(Attributes input)
{
    Varyings output = (Varyings)0;

    UNITY_SETUP_INSTANCE_ID(input);
    UNITY_TRANSFER_INSTANCE_ID(input, output);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(output);

    // It is important to place this function right after builtin 'instanceID'
initialization
    ComputeDeformedVertex_float(input.vertexID, input.vertex.xyz, input.normal.xyz,
input.tangent.xyz,
```

```
                                input.vertex.xyz, input.normal.xyz, input.tangent.xyz);
    ...
}
```

If the vertex attribute structure does not include `normal` or `tangent` members, replace them with dummy variables defined before calling `ComputeDeformedVertex_float`:

```
struct MinimalAttributes
{
  float4 vertex         : POSITION;
  uint vertexID         : SV_VertexID;
  UNITY_VERTEX_INPUT_INSTANCE_ID
};

Varyings ShadowPassVertex(MinimalAttributes input)
{
    Varyings output = (Varyings)0;

    UNITY_SETUP_INSTANCE_ID(input);
    UNITY_TRANSFER_INSTANCE_ID(input, output);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(output);

    float3 unusedNormal = 0;
    float3 unusedTangent = 0;

    ComputeDeformedVertex_float(input.vertexID, input.vertex.xyz, unusedNormal,
unusedTangent,
                                input.vertex.xyz, unusedNormal, unusedTanent);
    ...
}
```

Third-Party Shaders

Rukhanka Technologies can help you adapt custom shaders to work with **Rukhanka Animation**. Please contact us by support email or the official `Discord` server.

Deformation System

The `Rukhanka Deformation System` is a system that responsible for applying deformations calculated by the `Rukhanka Animation System` to the skinned meshes. These deformations include animations represented by skin poses and blend shapes represented by animated blend shape weights. It was made to be a direct replacement for `Entities.Graphics Deformation System` with following improvements:

- Each mesh vertex is skinned only with bones that have non-zero influence (weight) on it. This way redundant zero-weight loop iterations were completely removed from skinning function.
- Skinning and blend shape applications are performed by single compute shader.
- Deformations of all skinned meshes are performed by just two compute shader calls: per-vertex workload preparation and actual deformation application.
- The `Rukhanka Deformation System` works in cooperation with the `Rukhanka Animation Culling System` to check LOD level visibility, to restrict deformation application to the visible LOD levels only.
- The `Rukhanka Deformation System` works in cooperation with the `Rukhanka Animation Culling System` to apply deformations only to the visible meshes.

Animator Parameters

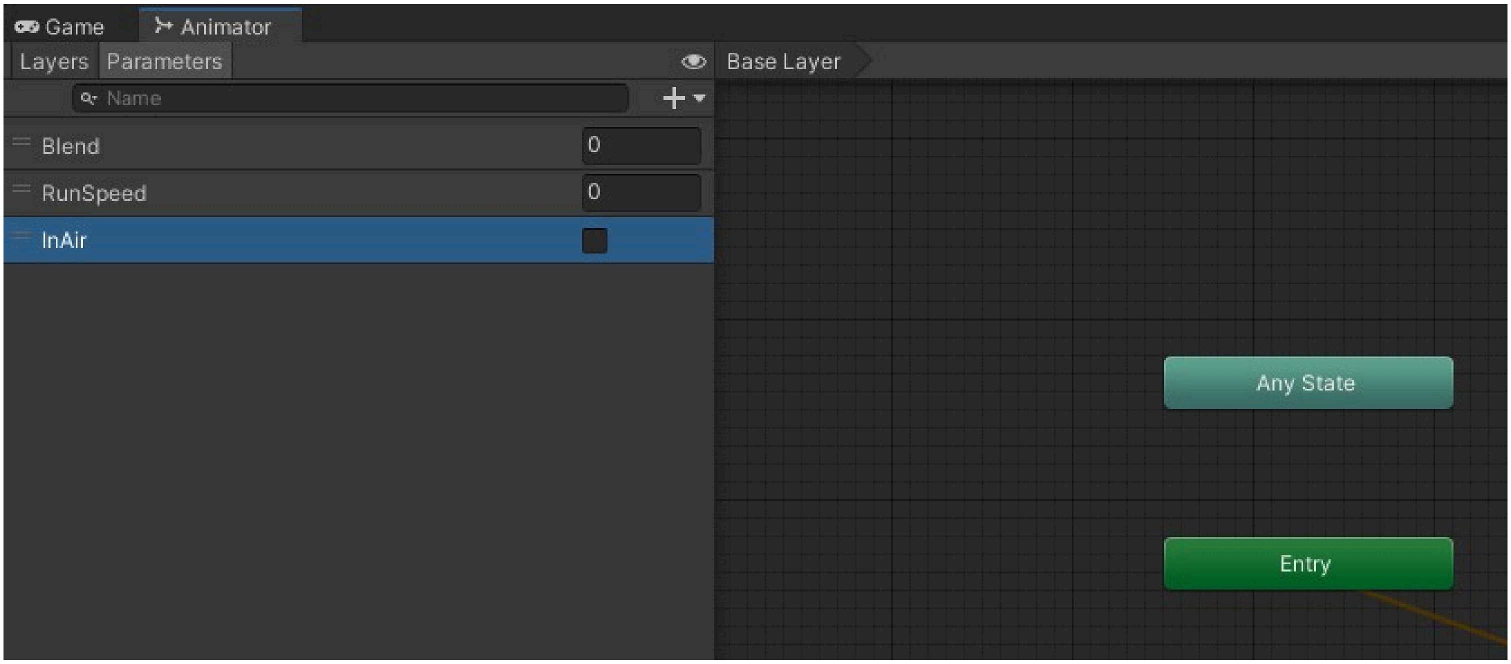
Direct Buffer Indexing

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":



Accessing by Hash

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, the helper class named `FastAnimatorParameter` and `AnimatorParametersAspect` aspect were 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 `AnimatorParametersAspect` and use the `FastAnimatorParameter` methods to access parameter value:

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

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

    void Execute(DynamicBuffer<AnimatorControllerParameterComponent>
animatorParametersBuf,
                 AnimatorControllerParameterIndexTableComponent
animatorParametersIndexTable)
    {
        var paramAspect = new AnimatorParametersAspect(animatorParametersBuf,
animatorParametersIndexTable);
        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

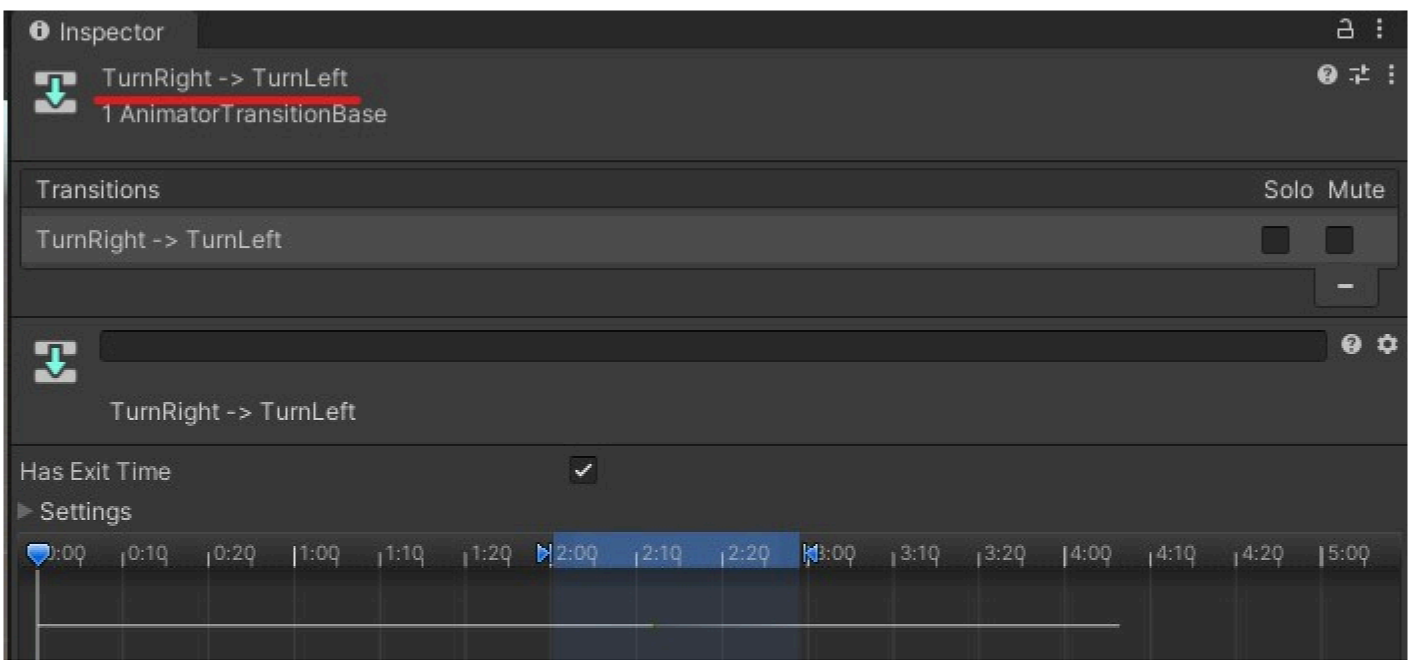
It is possible to query the runtime animator's internal state using the `AnimatorStateQueryAspect` aspect. State and transition data returned by this aspect contain a hash of state/transition respectively and normalized time (how much time controller is in this state/transition). To find out the required 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:

```
using FixedStringName = Unity.Collections.FixedString512Bytes;
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`:

```
using FixedStringName = Unity.Collections.FixedString512Bytes;

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 layers in
SystemAPI.Query<DynamicBuffer<AnimatorControllerLayerComponent>>())
        {
            animatorState = new AnimatorStateQueryAspect(layers);

            //  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.

Altering Animation Results

CPU Animator GPU Animator

Rukhanka animation calculation engine works as two-phase process:

1. The animation **calculation** phase is performed by `AnimationProcessSystem`.
2. The animation **application** phase is performed by `AnimationApplicationSystem`.

Between the execution of these two systems, there is a point when animations are already calculated, but not applied to destination rigs yet. At this point any modifications of result animation data are possible, and performed changes will be applied to animated skeletons.

For convenience, a `ComponentSystemGroup` named `RukhankaAnimationInjectionSystemGroup` was created, and placed at this execution point. All systems willing to modify animation results should use the `[UpdateInGroup(typeof(RukhankaAnimationInjectionSystemGroup))]` attribute at their declaration.

Bone Data Modification System

To show how animation data can be modified, we will make a simple modification system. Our goal will be to change the world position of specific bone with data contained in another component.

1. Suppose, there is a component with required data:

```
struct BonePositionOverrideComponent: IComponentData
{
    public int boneIndex;       // We want to modify bone identified by this index
    public float3 newWorldPose; // World position that should be given to target bone
}
```

2. Create a system with proper declaration:

```
[UpdateInGroup(typeof(RukhankaAnimationInjectionSystemGroup))]
[RequireMatchingQueriesForUpdate]
public partial struct SimpleBonePositionOverride: ISystem
{
```

```
    ...
}
```

3. Processing job needs operate on bone transform data. We will get it before job initialization, in `OnUpdate` function:

```
public void OnUpdate(ref SystemState ss)
{
    ...
    ref var runtimeData = ref SystemAPI.GetSingletonRW<RuntimeAnimationData>().ValueRW;
    ...
}
```

4. To access bone transform data of specific rig, a helper `AnimationStream` structure was introduced:

```
void Execute(Entity entity, RigDefinitionComponent rigDef) // Execute function of
IJobEntity
{
    ...
    using var animStream = AnimationStream.Create(runtimeData, entity, rigDef);
    ...
}
```

5. `AnimationStream` has functions to get and set the world and local poses of every animated rig bone. We change signature of `IJobEntity.Execute` function to get `BonePositionOverrideComponent` data, which is used as source for bone transform modifications:

```
void Execute(Entity entity, RigDefinitionComponent rigDef, BonePositionOverrideComponent
bonePosOverride)
{
    ...
    animStream.SetWorldPosition(bonePosOverride.boneIndex, bonePosOverride.newWorldPose);
    ...
}
```

6. `Rukhanka` has delayed mechanism of all children's bone poses recalculation. During `Get` calls required bone position data will be recalculated automatically. Any outdated bone data that is left at the end of the `AnimationStream` object lifetime will be refreshed during the `Dispose()` call (or use `using` statement);

Final system and work job will look like:

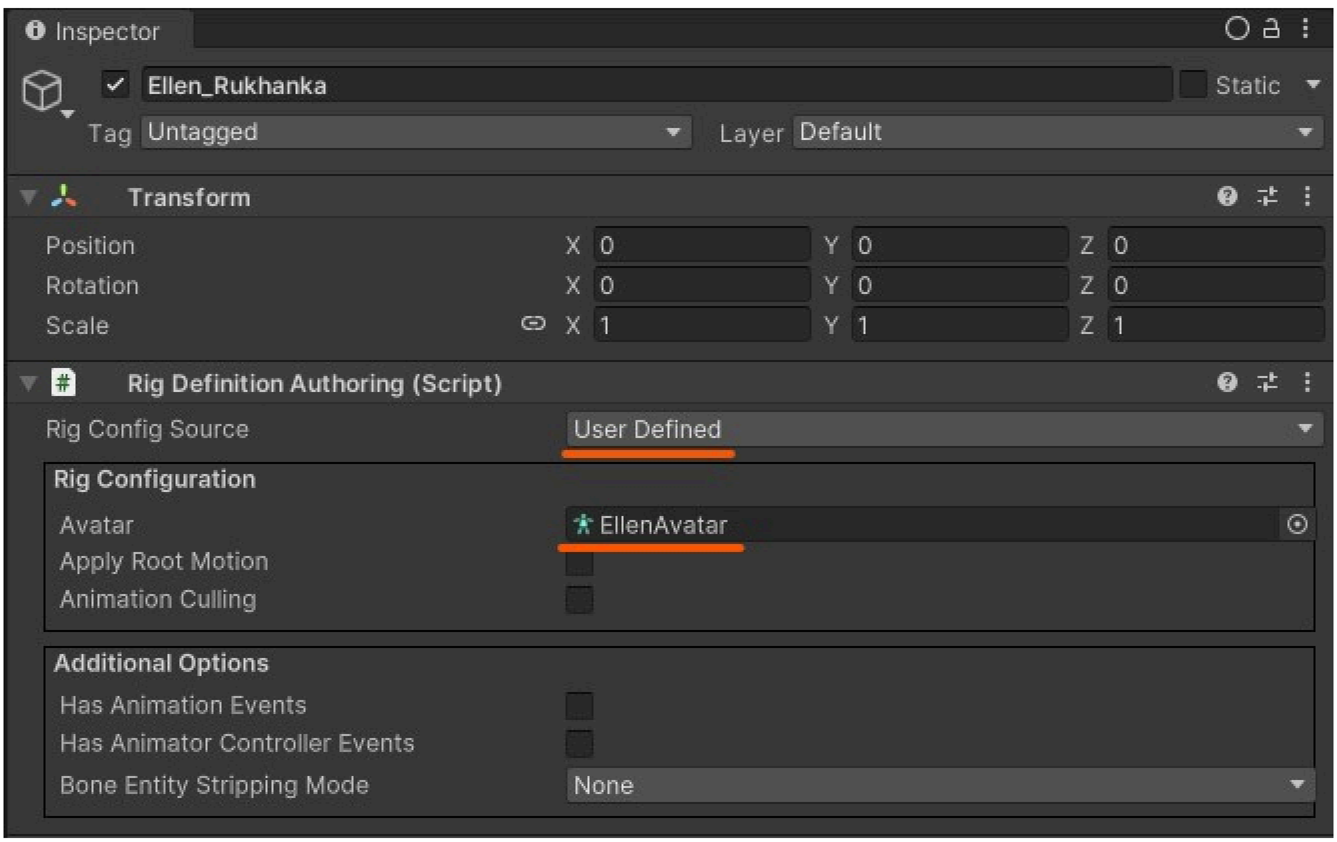
```
struct BonePositionOverrideComponent: IComponentData
{
    public int boneIndex;
    public float3 newWorldPose;
}

/////////////////////////////////////////////////////////////////////////////////

[UpdateInGroup(typeof(RukhankaAnimationInjectionSystemGroup))]
[RequireMatchingQueriesForUpdate]
public partial struct SimpleBonePositionOverride: ISystem
{
    [BurstCompile]
    partial struct BonePositionOverrideJob : IJobEntity
    {
        [NativeDisableContainerSafetyRestriction]
        public RuntimeAnimationData runtimeData;

        void Execute(Entity entity, RigDefinitionComponent rigDef,
BonePositionOverrideComponent bonePosOverride)
        {
            using var animStream = AnimationStream.Create(runtimeData, entity, rigDef);
            animStream.SetWorldPosition(bonePosOverride.boneIndex,
bonePosOverride.newWorldPose);
            // Implicit animationStream.Dispose() call is here
        }
    }

/////////////////////////////////////////////////////////////////////////////////

    [BurstCompile]
    public void OnUpdate(ref SystemState ss)
    {
        ref var runtimeData = ref SystemAPI.GetSingletonRW<RuntimeAnimationData>
().ValueRW;
        var job = new BonePositionOverrideJob()
        {
            runtimeData = runtimeData,
        };

        job.ScheduleParallel();
    }
}
```

Working without Animator

`AnimatorControllerSystem`, baked from Unity's `Animator` component, orchestrates a high level of animation processing computation. In a nutshell, it defines what animation needs to be played at a given point in time and calculates animation clip weight for all relevant animations. Processing flow can be controlled by using animator parameters, but this is roughly the only way to configure animator behavior. Every project has a distinct set of animator controller requirements, and **Rukhanka** can work with custom user-written controller systems.

Authoring Preparation

`Rig Definition Authoring` should be switched into `User Defined` rig config source mode. Normally **Rukhanka** reads all required rig information from Unity's `Avatar` component. Animated model `Avatar` needs to be specified explicitly in user mode.



Now `Animator` component can be removed from authoring `GameObject`.

Scripted Animator

Without authoring `Animator` there is no `AnimatorControllerLayerComponent` buffer created for animated entity. Therefore `AnimatorControllerSystem` will no longer orchestrate animation flow for this entity. Now it is your, as a user of **Rukhanka**, task to command which animation should be played.

Components and Baking

Assume that we want to play a single animation in a loop. Let's define the required authoring and ECS components:

```
using Unity.Entities;
using UnityEngine;

//  Authoring UnityEngine.Component
public class MyAnimatorAuthoring: MonoBehaviour
{
    //  Clip to play
    public AnimationClip clip;
}

//  Runtime ECS component
public struct MyAnimatorComponent: IComponentData
{
    //  Current animation time that will be advanced by custom controller system
    public float normalizedAnimationTime;
    //  Animation clip blob that Rukhanka bakes from authoring UnityEngine.AnimationClip
    public BlobAssetReference<Rukhanka.AnimationClipBlob> clipBlob;
}
```

The baker for `MyAnimatorComponent` should instruct **Rukhanka** to bake the given animation clip. Result `BlobAssetReference` needs to be stored in `MyAnimatorComponent.clipBlob` field.

```
//  Baker for the MyAnimatorAuthoring
class MyAnimatorBaker: Baker<MyAnimatorAuthoring>
{
    public override void Bake(MyAnimatorAuthoring authoring)
    {
        //  Avatar is needed for baking purposes
        var rigDef = GetComponent<Rukhanka.Hybrid.RigDefinitionAuthoring>();
        var avatar = rigDef.GetAvatar();
```

```
        // Instantiate AnimationClipBaker class and bake required animations
        var animationBaker = new Rukhanka.Hybrid.AnimationClipBaker();
        var bakedAnimations = animationBaker.BakeAnimations(this, new [] {authoring.clip},
avatar, authoring.gameObject);

        //  Baked animation should be stored in the animation blob database for the
ability to be queried in runtime using animation hash
        //  NewBlobAssetDatabaseRecord buffer serves this purpose
        var entity = GetEntity(authoring, TransformUsageFlags.None);
        var newBlobsBuffer =
AddBuffer<Rukhanka.NewBlobAssetDatabaseRecord<Rukhanka.AnimationClipBlob>>(entity);
        var animationBlobAssetReference = bakedAnimations[0];
        if (animationBlobAssetReference.IsCreated)
        {
            var newAnimationBlob = new
Rukhanka.NewBlobAssetDatabaseRecord<Rukhanka.AnimationClipBlob>()
            {
                hash = animationBlobAssetReference.Value.hash,
                value = animationBlobAssetReference
            };
            newBlobsBuffer.Add(newAnimationBlob);
        }

        //  Construct MyAnimatorComponent and assign baked animation clip blob asset
reference to it
        var myAnimator = new MyAnimatorComponent()
        {
            clipBlob = animationBlobAssetReference,
            normalizedAnimationTime = 0
        };
        AddComponent(entity, myAnimator);
    }
}
```

Animator Controller System

A simple scripted controller system needs to perform two tasks:

- Increase `MyAnimatorComponent.normalizedAnimationTime` value according to time flow.
- Update `Rukhanka.AnimatioToProcessComponent` buffer to instruct **Rukhanka** to play specific animation at a given time.

```
using Unity.Burst;
using Unity.Entities;

[UpdateBefore(typeof(Rukhanka.RukhankaAnimationSystemGroup))]
partial struct MyAnimatorSystem: ISystem
{
    [BurstCompile]
    partial struct MyAnimatorControllerJob: IJobEntity
    {
        public float deltaTime;
        void Execute(ref MyAnimatorComponent myAnimator, ref
DynamicBuffer<Rukhanka.AnimationToProcessComponent> animationToProcessBuffer)
        {
            //  Increase animation play time by deltaTime. Take into account animation
length in seconds
            myAnimator.normalizedAnimationTime += deltaTime /
myAnimator.clipBlob.Value.length;
            //  Clear all previous frame animations
            Rukhanka.ScriptedAnimator.ResetAnimationState(ref animationToProcessBuffer);
            //  Play animation at a given time
            Rukhanka.ScriptedAnimator.PlayAnimation(ref animationToProcessBuffer,
myAnimator.clipBlob, myAnimator.normalizedAnimationTime);
        }
    }

    [BurstCompile]
    public void OnUpdate(ref SystemState ss)
    {
        var myAnimatorControllerJob = new MyAnimatorControllerJob()
        {
            deltaTime = SystemAPI.Time.DeltaTime
        };
        myAnimatorControllerJob.ScheduleParallel();
    }
}
```

This is all that is needed to play animation with **Rukhanka**. `ScriptedAnimator` class provides helper functions to play and blend animations.

Mixed Controller Mode

Often it is suitable to define necessary states using the `Macanim` animator design window, but control state transitions manually from own code. With **Rukhanka** you can use `ScriptedAnimator.PlayAnimatorState` function to play motion defined in the state (single animation, blend tree, or even blend tree of blend trees). In this mode, you must keep authoring `Animator` component with a given controller on the animated object.

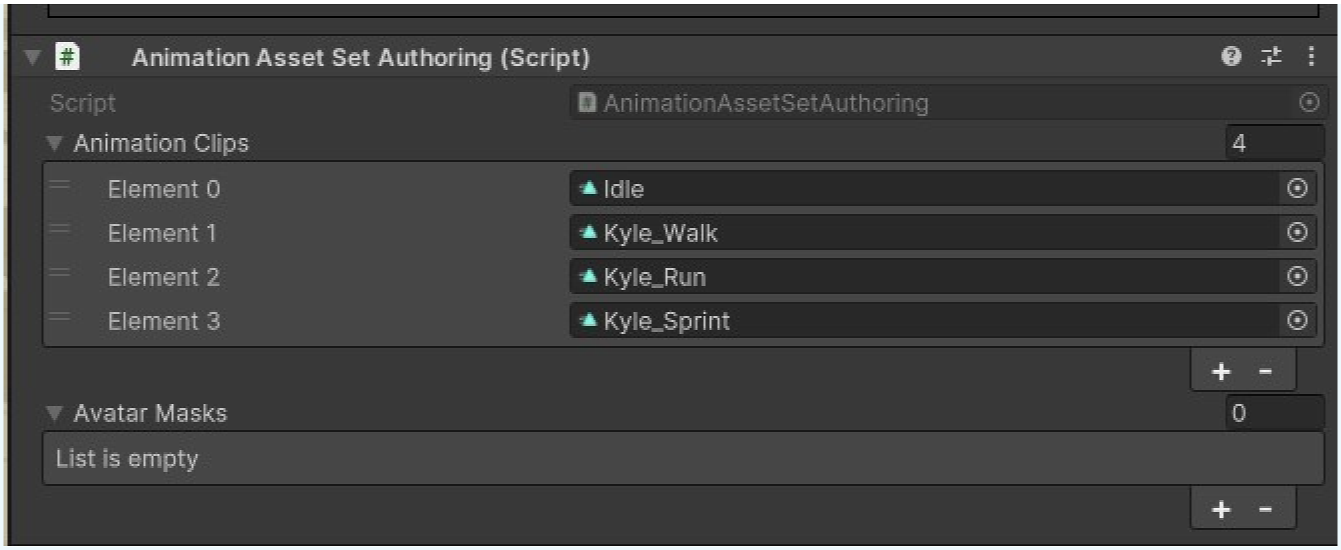
> **WARNING**
>
> Do not forget to disable `FillAnimationsFromControllerSystem` and `AnimatorControllerSystem` systems because your controller system will be either overridden or override the results of default **Rukhanka's** controller systems.

Blob Database

Sometimes, elsewhere baked animation clip blob need to be runtime accessed. To serve this purpose `BlobDatabaseSingleton` was introduced. Store only animation hash during the baking stage, and query animation hash map during runtime:

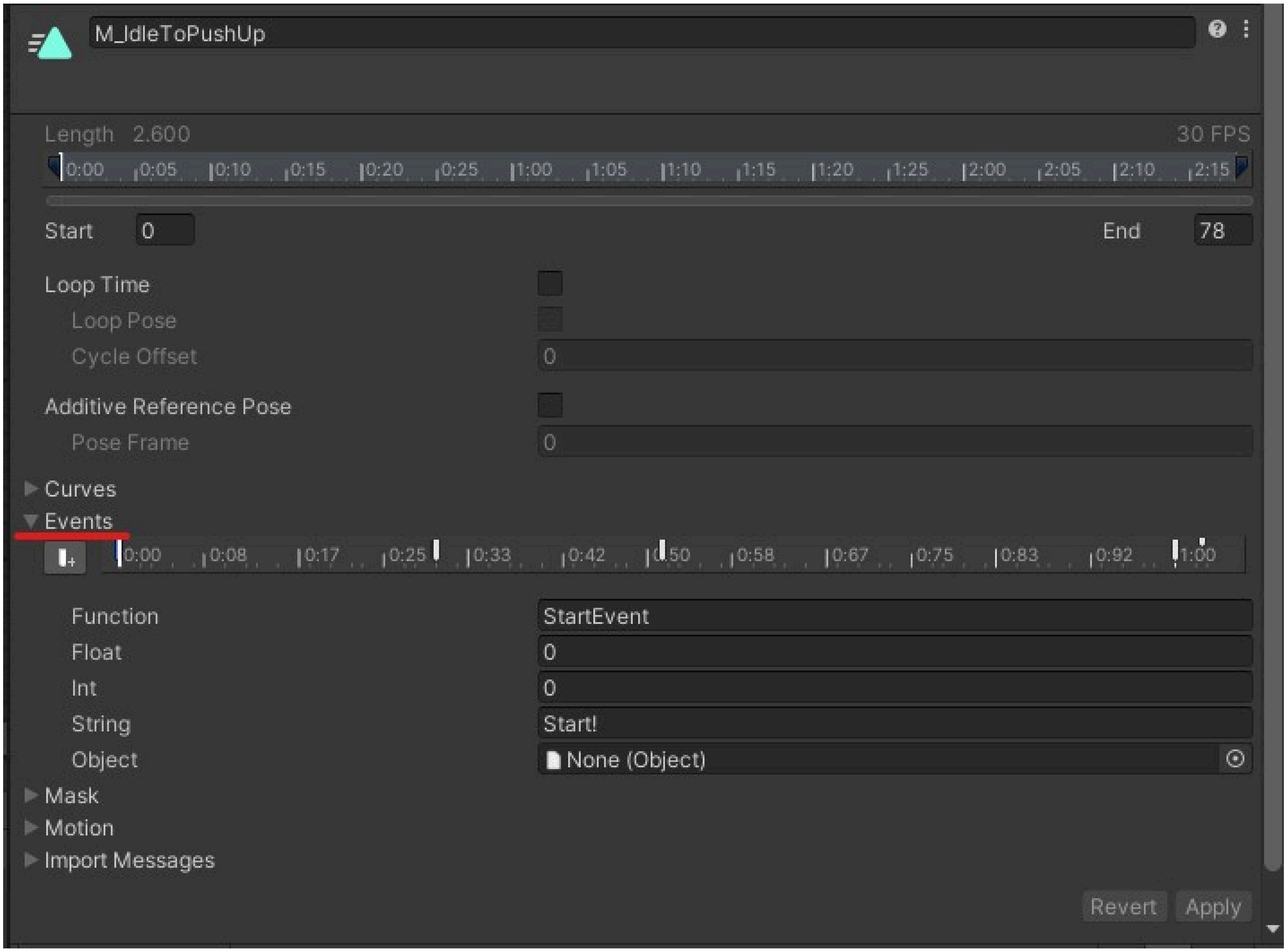
```
//  Baker
AnimationClip authoringClip = ...
Avatar authoringAvatar = ...
var animationHash = Rukhanka.Hybrid.BakingUtils.ComputeAnimationHash(authoringClip,
authoringAvatar);

...

// SystemBase or ISystem
if (SystemAPI.TryGetSingleton<Rukhanka.BlobDatabaseSingleton>(out var blobDBSingleton))
{
    blobDBSingleton.animations.TryGetValue(animationHash, out var
animationClipBlobAssetReference);
}
```

> **INFO**
>
> There is a handy authoring component named `AnimationAssetSetAuthoring` that can be used to simplify animation clip and avatar mask blob assets baking.


Animation Asset Set Authoring (Script)
Script
AnimationAssetSetAuthoring
Animation Clips
4
Element 0
Idle
Element 1
Kyle_Walk
Element 2
Kyle_Run
Element 3
Kyle_Sprint
Avatar Masks
0
List is empty

Animation Events

Animation events are implemented in the form of a component buffer that is filled by the animation processing engine. Events should be defined in Unity's animation configuration dialog:



`Animation Event` properties are translated into the following component:

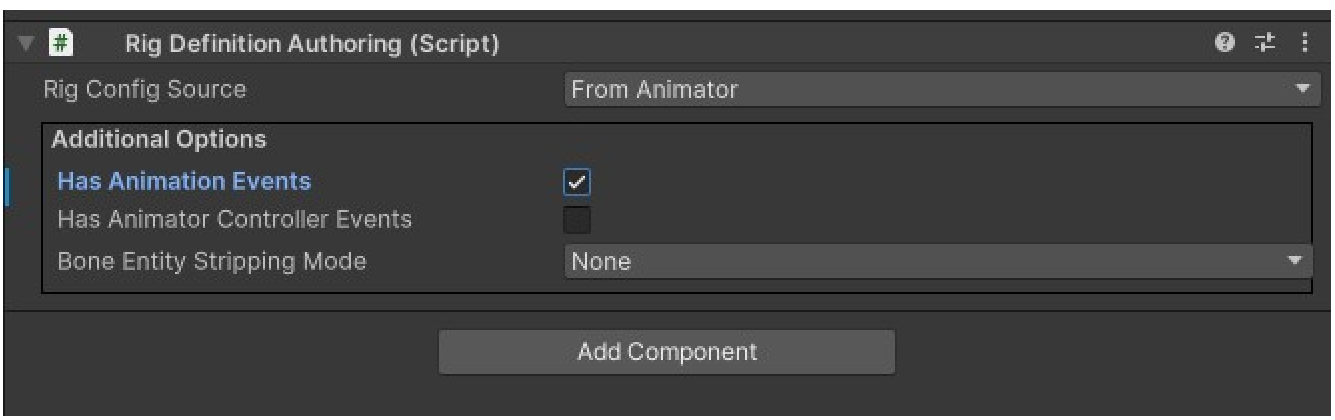
```
public struct AnimationEventComponent: IBufferElementData, IEnableableComponent
{
    public uint nameHash;
    public float floatParam;
    public int intParam;
    public uint stringParamHash;
}
```

- **nameHash** - contains a hash value of Unity's animation event `Function` field. It is calculated by `FixedStringExtensions.CalculateHash32()` helper function.
- **floatParam** - contains authoring animation event `Float` field.
- **intParam** - contains authoring animation event `Int` field.
- **stringParamHash** - contains a hash value of authoring animation event `String` field. It is calculated by `FixedStringExtensions.CalculateHash32()` helper function. The `Object` field of the authoring event is ignored.

In every frame, **Rukhanka's** animation compute engine fills this buffer with events that occurred in this frame and clears all previous events. So events that occurred in the previous frame are available before `RukhankaAnimationSystemGroup`. Events occurred in the current frame available after `RukhankaAnimationSystemGroup`.

Enabling Animation Events

By default, the animation events component buffer is not added to the animated entity. `RigDefinitionAuthoring` has a checkbox named `Has Animation Events` which orders to add `AnimationEventComponent` buffer to the rig entity at the baking stage.



Animator Controller Events

Animator controller events are implemented in the form (just like animation events) of a component buffer that is filled by the animator state machine processing engine.

The animator controller event is a components buffer element with the following declaration:

```
public struct AnimatorControllerEventComponent : IBufferElementData, IEnableableComponent
{
    public enum EventType
    {
        StateEnter,
        StateExit,
        StateUpdate
    }

    public EventType eventType;
    public int layerId;
    public int stateId;
    public float timeInState;
}
```

Rukhanka's state machine compute engine will fill this buffer with events that occurred in the current processing frame and clear all events from the previous frame. State machine events exist in the following types:

- **StateEnter** - this event appears in the frame when the state machine enters some state.
- **StateExit** - this event appears in the frame when the state machine exits some state.
- **StateUpdate** - this event appears in the frames when the state machine advances through some state and there are no **StateEnter** and **StateExit** for such state.

These event types have a close relation to the `OnStateEnter`, `OnStateExit`, and `OnStateUpdate` of the `StateMachineBehaviour` Unity component.

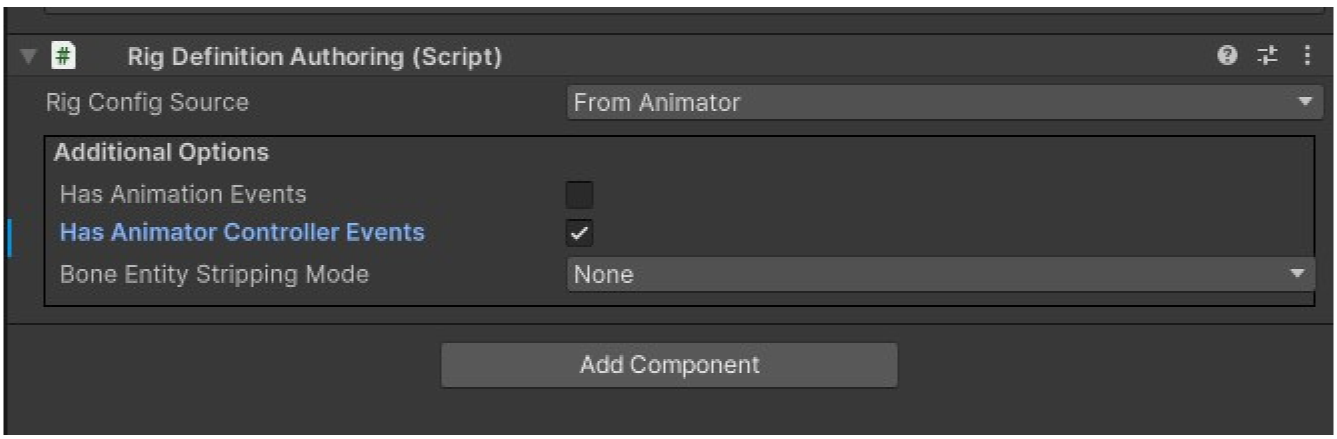
The component fields are:

- **eventType** - type of the event.

- **layerId** - index of the layer of baked animator controller. Layer data can be accessed by using this index in the `AnimatorControllerLayerComponent` buffer.
- **stateId** - index of the state of the baked animator controller layer. State data can be accessed by using this index in `LayerBlob.states` array of controller blob.
- **timeInState** - normalized state time. It has range from 0 to 1.

Enabling Animator Controller Events

By default, the animator controller events component buffer is not added to the animated entity. `RigDefinitionAuthoring` has a checkbox named `Has Animator Controller Events` which orders to add `AnimatorControllerEventComponent` buffer to the rig entity at the baking stage.



Inverse Kinematics

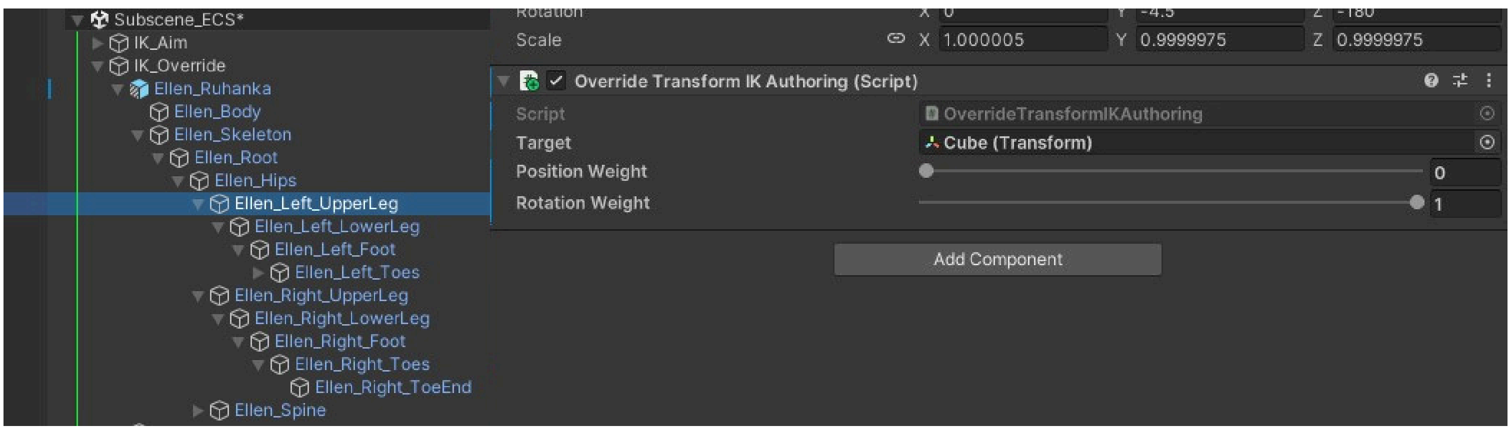
CPU Animator GPU Animator

With the ability to alter animation before it is applied to the resulting rig, there is a possibility of **Inverse Kinematics (IK)** algorithms in **Rukhanka** becoming available.

There are several IK algorithms available in the base **Rukhanka** distribution:

Override Transform

This is a very simple algorithm, which for a given rig bone overrides position and/or rotation taken from other entity. It can be used by attaching `OverrideTransformIKAuthoring` authoring `MonoBehaviour` to the bone entity that needs to be affected:



The following parameters can be configured:

- **Target** - an entity whose position and rotation will be used for affected bone.
- **Position Weight** - degree of influence of target position on result bone placement.
- **Rotation Weight** - degree of influence of target rotation on result bone placement.

> **INFO**
>
> Weight values for all algorithms are ranged between 0 and 1. If zero, the algorithm will have no influence on the final pose. If one, the bone pose will be completely overridden by the algorithm

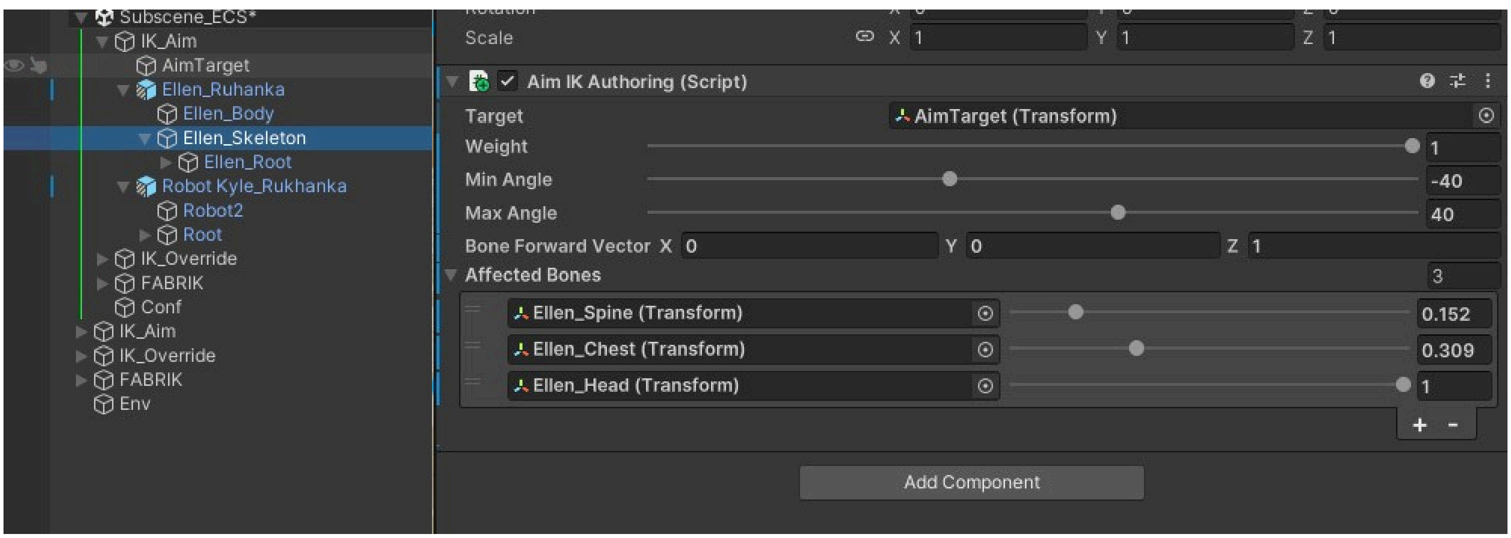
output pose. Values in between will interpolate linearly between the original bone pose and algorithm output.

Corresponding `OverrideTransformIKComponent` has the following declaration:

```
public struct OverrideTransformIKComponent: IComponentData, IEnableableComponent
{
    public Entity target;
    public float positionWeight;
    public float rotationWeight;
}
```

Aim

The aim algorithm rotates a bone chain to face a target entity. It can be enabled and configured by using `AimIKAuthoring` authoring `MonoBehaviour` to any bone of the affected rig:



Configuration parameters:

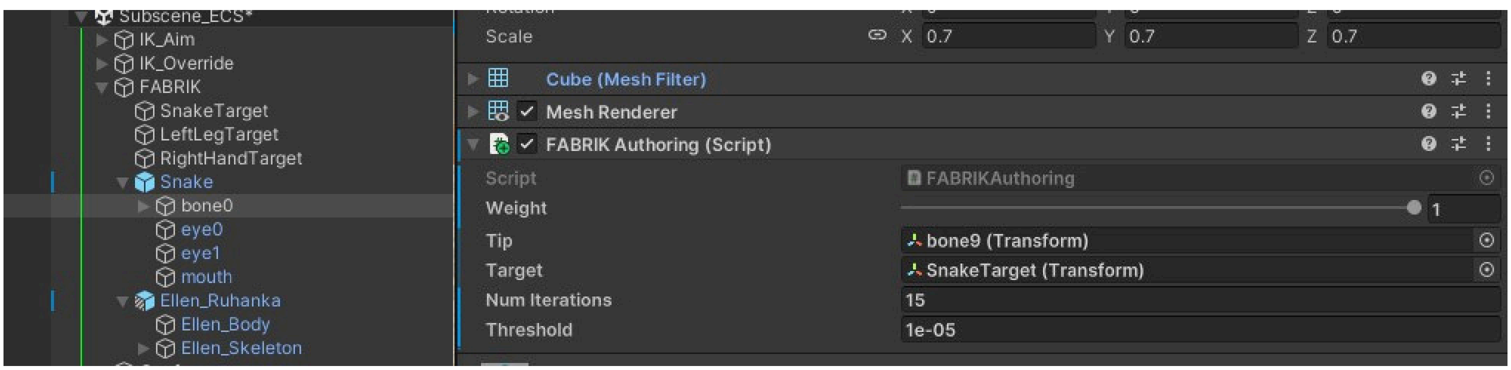
- **Target** - affected bones will be rotated to face this entity.
- **Weight** - degree of influence of aimed rotation on result bones orientation.
- **Min and Max Angles** - aimed rotation can be constrained by minimum and maximum angles (given in degrees).
- **Bone Forward Vector** - a vector that will be used by the aim algorithm as a forward vector.
- **Affected Bones** - list bones with associated weights. Bones will be processed in the order they are declared. To prevent multiple rotations make sure that parent bones come before children.

`AimIKAuthoring` converted to `AimIKComponent` with base algorithm parameters and `AimIKAffectedBoneComponent` buffer with affected bones list. They have the following declarations:

```
public struct AimIKComponent: IComponentData, IEnableableComponent
{
    public Entity target;
    public float2 angleLimits;
    public float3 forwardVector;
    public float weight;
}

public struct AimIKAffectedBoneComponent : IBufferElementData
{
    public Entity boneEntity;
    public float weight;
}
```

Forward And Backward Reaching Inverse Kinematics (FABRIK)

FABRIK algorithm can be used when a chain of connected bones needs to reach some target position. `FABRIKAuthouring` authoring `MonoBehaviour` can be used to enable and configure the algorithm. `FABRIKAuthouring` should be attached to the root bone of the affected chain.



It has the following configurable properties:

- **Weight** - degree of influence of the FABRIK algorithm on positions and rotations of the affected bone chain.
- **Tip** - end of the affected bone chain. All bones from the root chain bone (the bone with `FABRIKAuthouring` attached) and this bone will be affected by the FABRIK algorithm.
- **Target** - an entity whose pose will be used as a goal position.
- **Num Iterations** - maximum number of algorithm iterations in an attempt to reach the target.

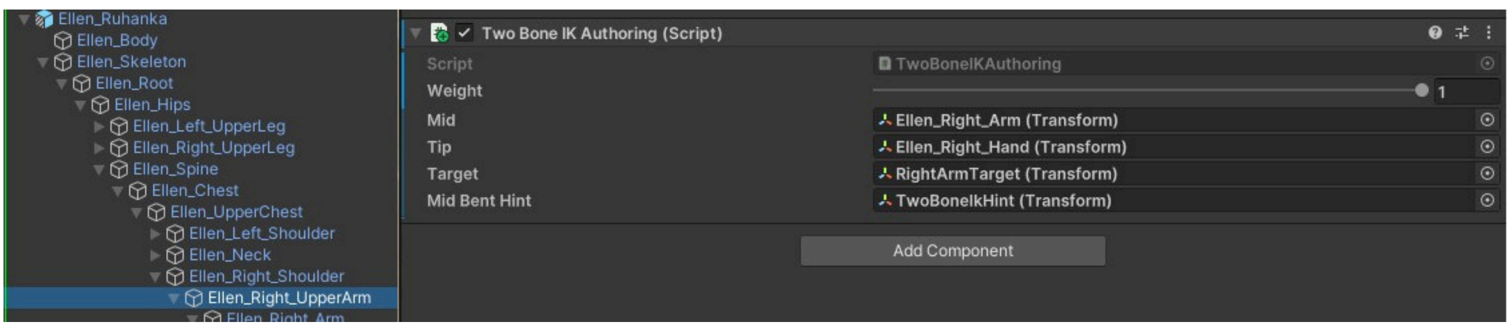
- **Threshold** - chain tip and target position difference value that will be treated by an algorithm as "acceptable" during solving. If the tip-to-target distance is less than the **threshold** FABRIK will stop its computation.

`FABRIKAuthouring` is converted into `FABRIKComponent` with the following declaration:

```
public struct FABRIKComponent: IComponentData, IEnableableComponent
{
    public Entity tip, target;
    public int numIterations;
    public float weight, threshold;
}
```

Two Bone IK

`Two Bone IK` is a simple algorithm that operates on two connected bones. It is used for reaching the target position by two-bone systems like lower-upper legs/arms. `TwoBoneIKAuthoring` should be attached to the root bone of the affected chain.



It has the following configurable properties:

- **Weight** - degree of influence of the IK algorithm on rotations of the affected bones.
- **Mid** - middle bone of the affected chain.
- **Tip** - end of the affected bone chain.
- **Target** - an entity whose pose will be used as a goal position.
- **Mid Bend Hint** - an entity whose pose will be used to correctly orient middle bone bend orientation. Generally, it should be located in front of the desired mid-bone position.

`TwoBoneIKAuthouring` is converted into `TwoBoneIKComponent` with the following declaration:

```
public struct TwoBoneIKComponent: IComponentData, IEnableableComponent
{
```

```
    public Entity mid, tip, target, midBentHint;
    public float weight;
}
```

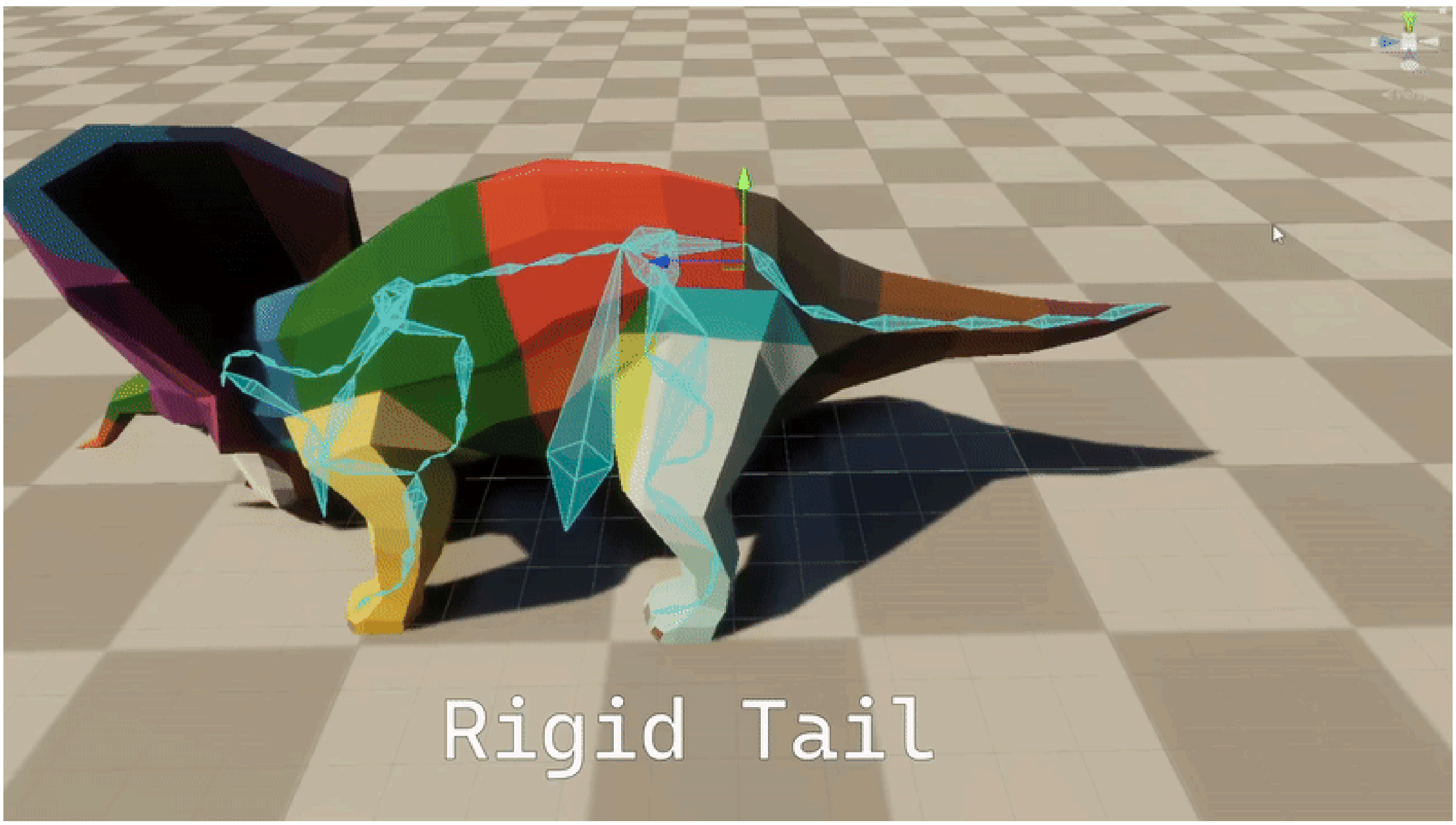
> **TIP**
>
> All described algorithms have a usage example in **Rukhanka** samples

Procedural Animation

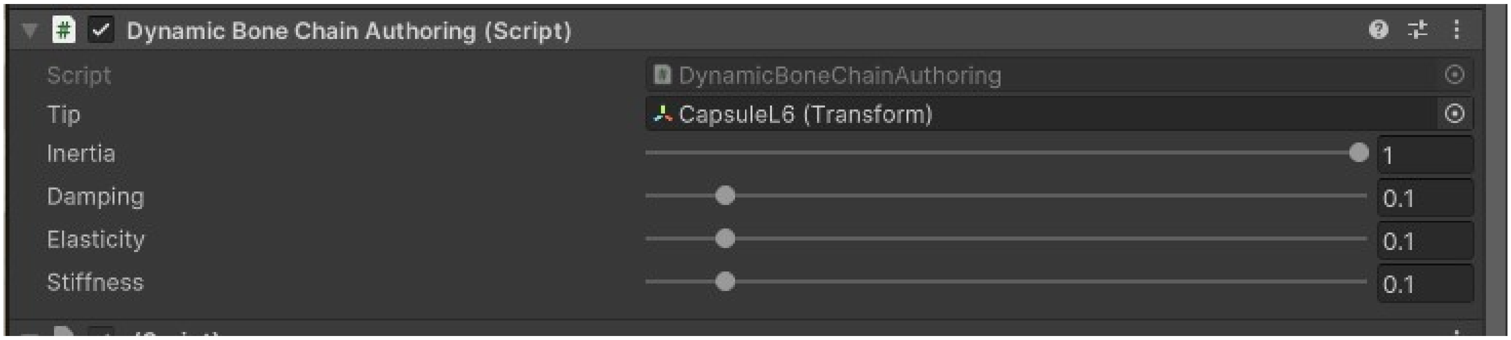
CPU Animator GPU Animator

The default animation processing path is to use animation clips to drive animated bone poses. Procedural animation is another approach to moving skeletal bone poses according to certain mathematical algorithms. Such algorithms should be implemented as systems that run within the `AnimationInjectionSystemGroup`.

Dynamic Bone Chain



The dynamic bone chain is a simulation system that applies spring-based physics, smoothing, and blending to transform hierarchies.



The authoring script must be placed on the root bone of the simulated hierarchy. The following parameters can be configured:

- **Tip** - the end bone of the simulated bone chain.
- **Damping** - controls how quickly a dynamic bone's oscillations lose energy, preventing endless wobbling by reducing velocity each frame.
- **Elasticity** - determines how strongly a dynamic bone tries to return to its rest position, acting like the stiffness of a spring.
- **Stiffness** - controls how rigidly a dynamic bone resists bending, keeping it closer to its original orientation instead of freely swaying.

The dynamic bone chain consists of one component and one buffer:

```
public struct DynamicBoneChainComponent: IComponentData, IEnableableComponent
{
    public float inertia;
    public float damping;
    public float elasticity;
    public float stiffness;
    public float timeAccumulator;
    public float3 prevPosition;
}

public struct DynamicBoneChainNode: IBufferElementData
{
    public int parentIndex;
    public float3 position;
    public float3 prevPosition;
    public BoneTransform referenceLocalPose;
    public Entity boneEntity;
}
```

Non-skinned Mesh Animation

CPU Animator GPU Animator

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.

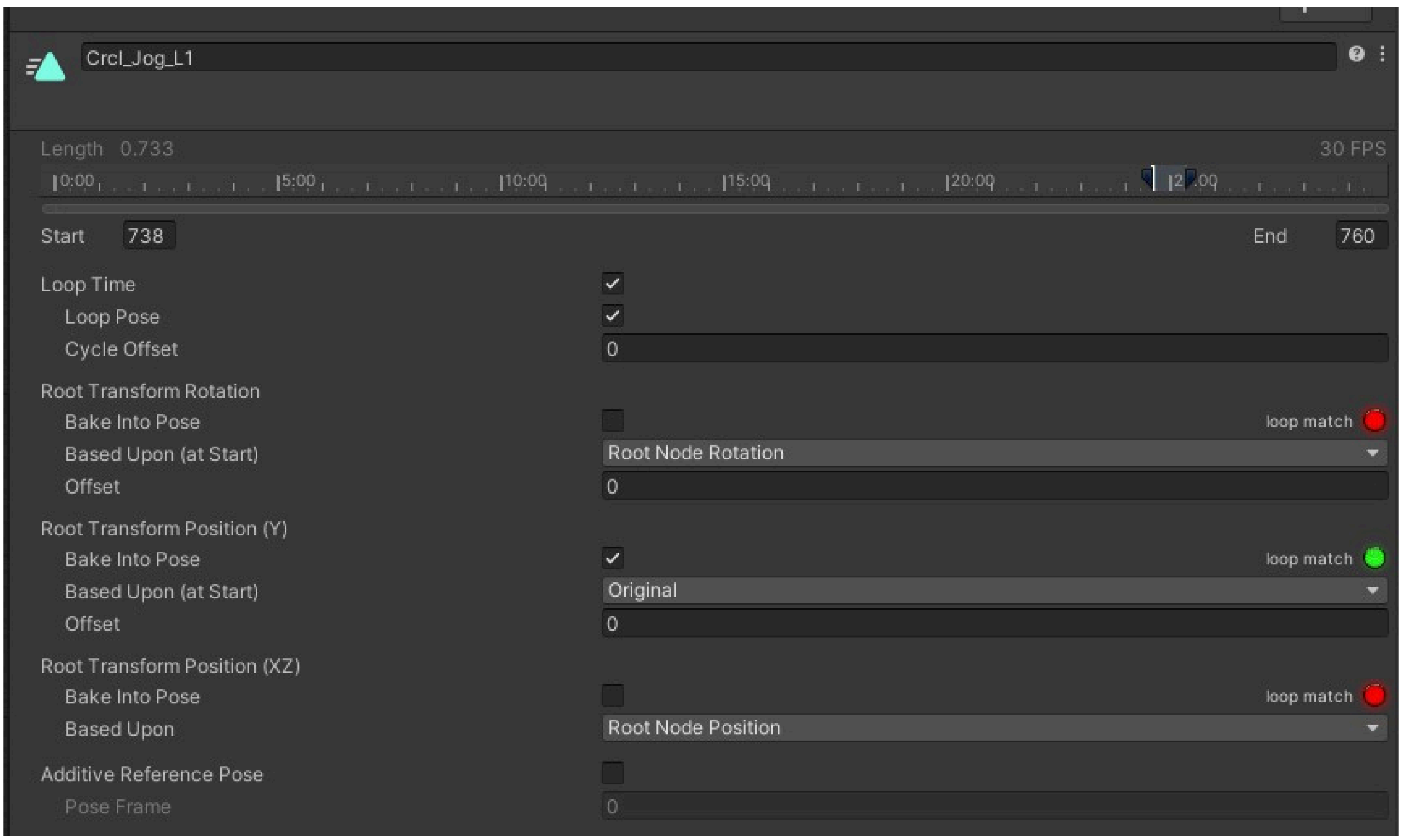
Root Motion

CPU Animator GPU Animator

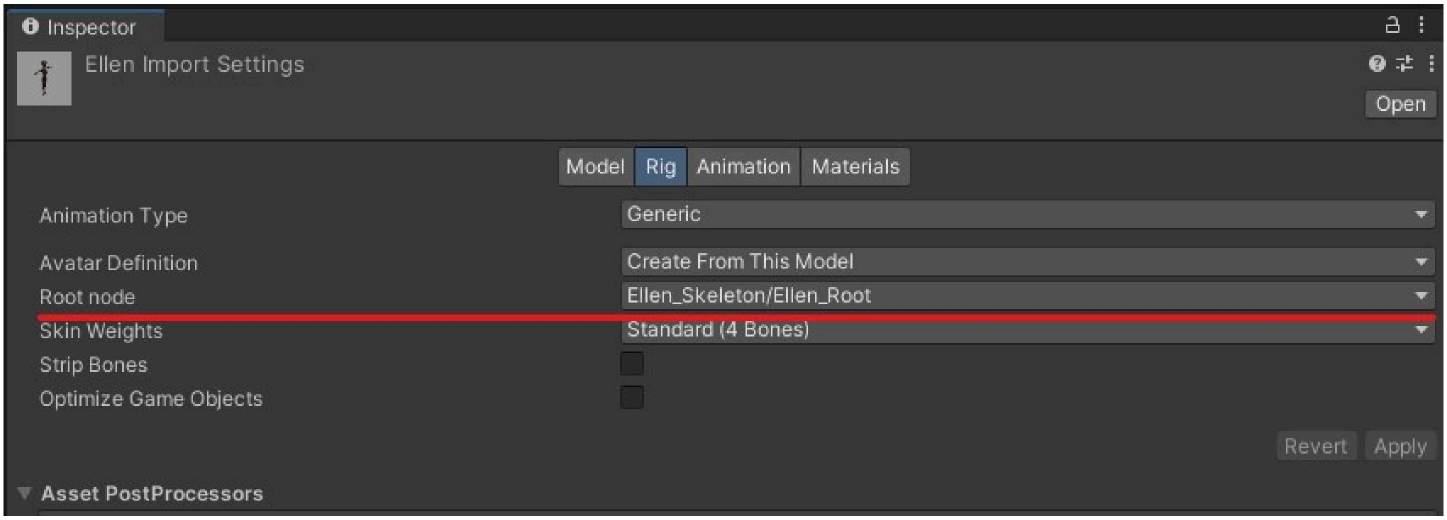
Rukhanka has full root motion support for humanoid and generic avatars and animations.

Follow these steps to enable root motion for your model:

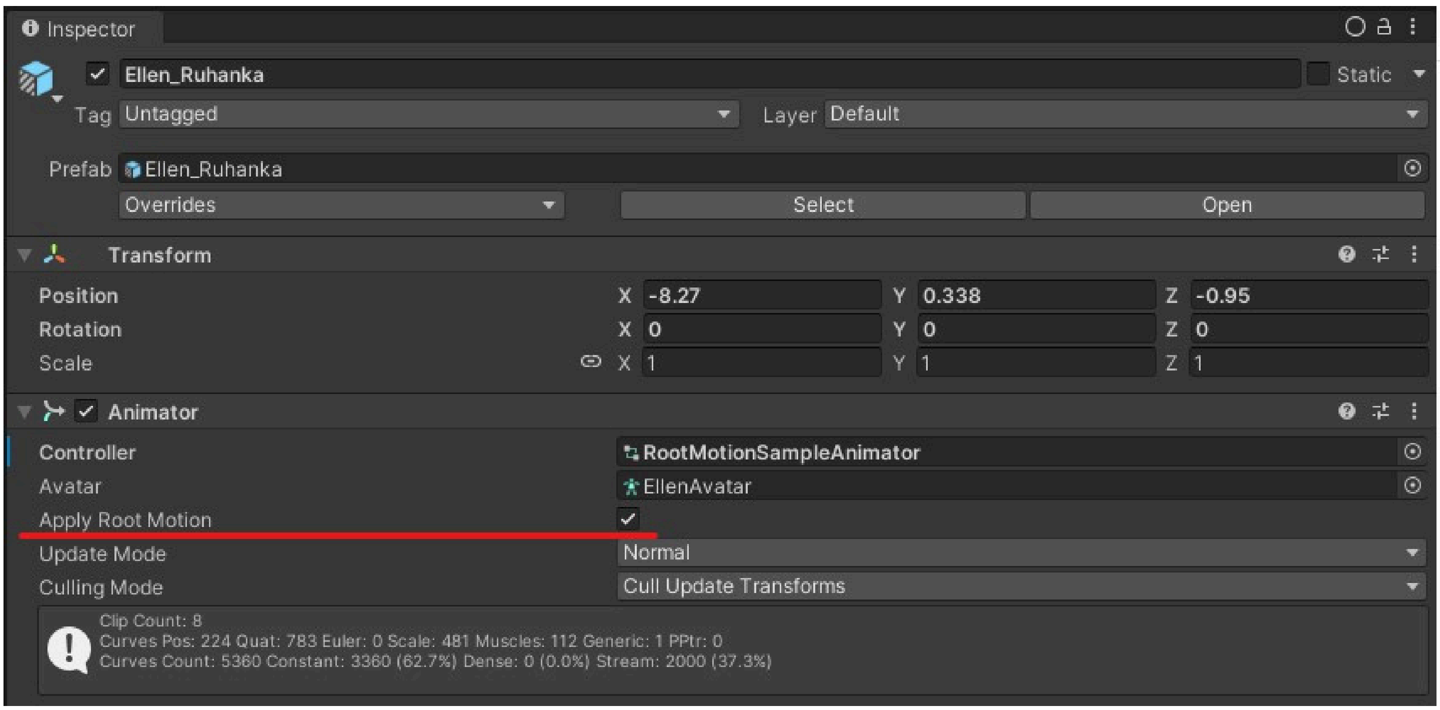
- Configure `Root motion` as usual from motion configuration section of animator importer window:



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



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

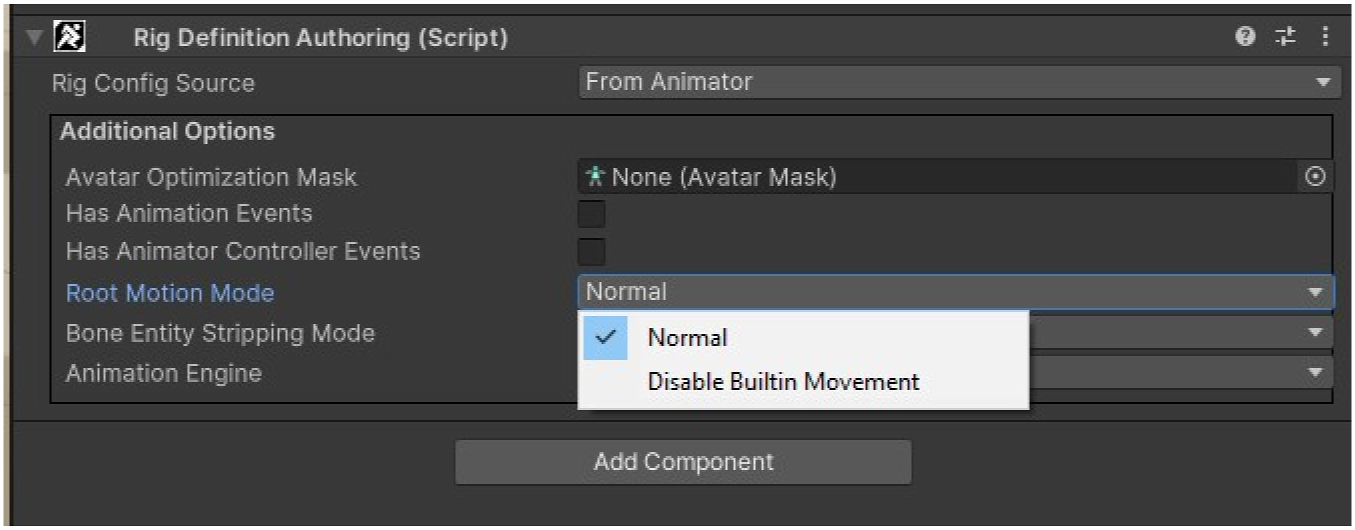


Custom root motion handling

Sometimes root motion calculated movement needs to be propagated to other systems. There is the `RootMotionVelocityComponent` that can be used to obtain root motion velocity for the current frame. If you don't want to move the animated entity by **Rukhanka's** root motion code, it can be disabled in `RigDefinitionAuthoring`. `Root Motion Mode` property controls **Rukhanka's** root motion operation mode:

- **Normal**. Standard operation mode. In this mode **Rukhanka** will move entity according to root motion tracks if this is requested.
- **Disable Builtin Movement**. Calculate root motion, but do not apply it to the entity position.

In both modes `RootMotionVelocityComponent` gets updated with current frame velocity values.



User Curves

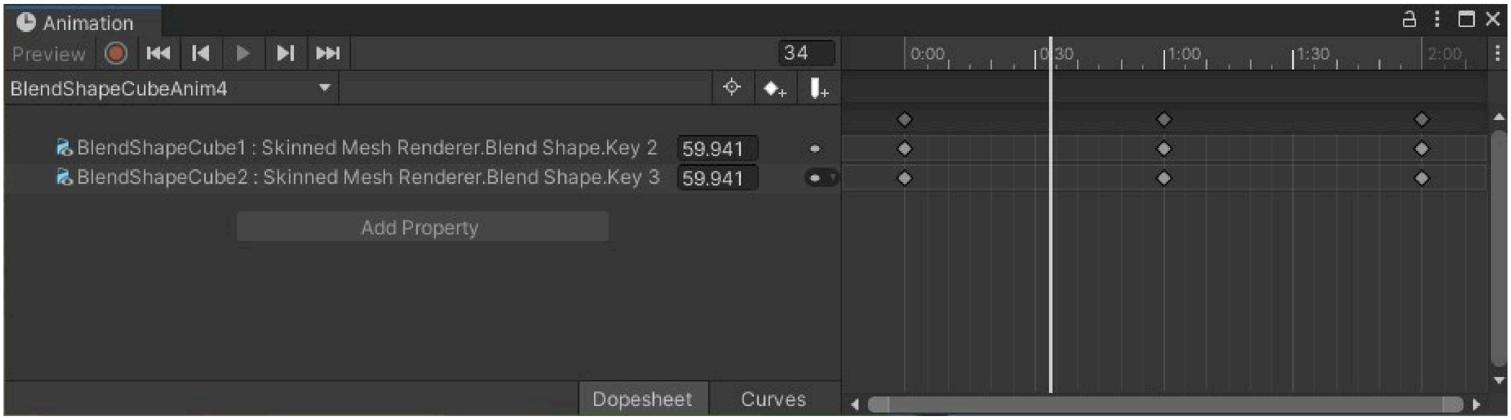
CPU Animator GPU Animator

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.

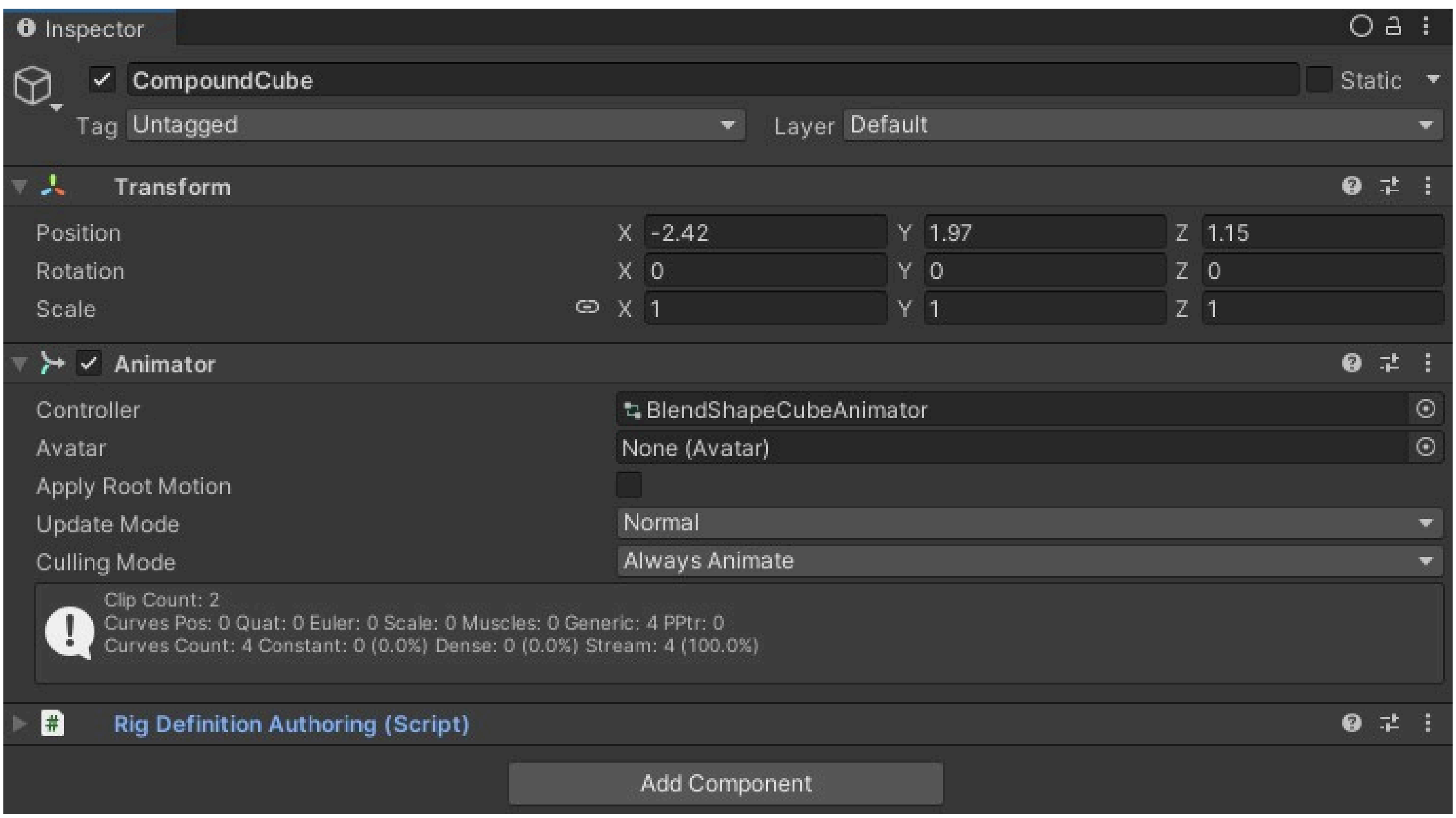
Blend Shapes

CPU Animator **GPU Animator**

- Prepare the model with blend shapes as described in the documentation.
- Make an animation that drives the blend shape key value of the skinned mesh render.



- Make an `Animator Controller` with prepared animations and set it to the authoring animator of an animated object.



WARNING

Make sure that skinned meshes use deformation-aware shader

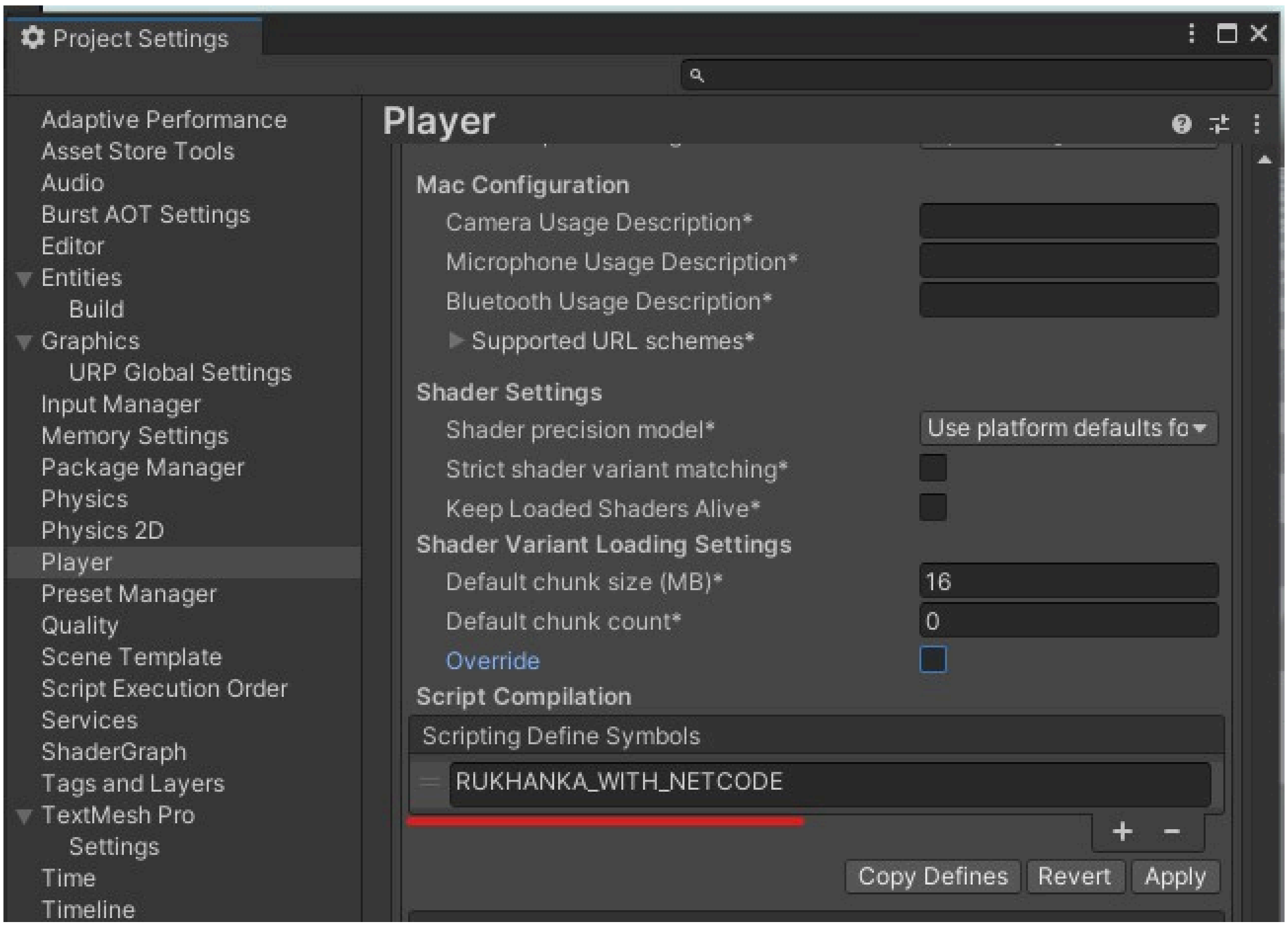
Bone Attachemnts

CPU Animator GPU Animator

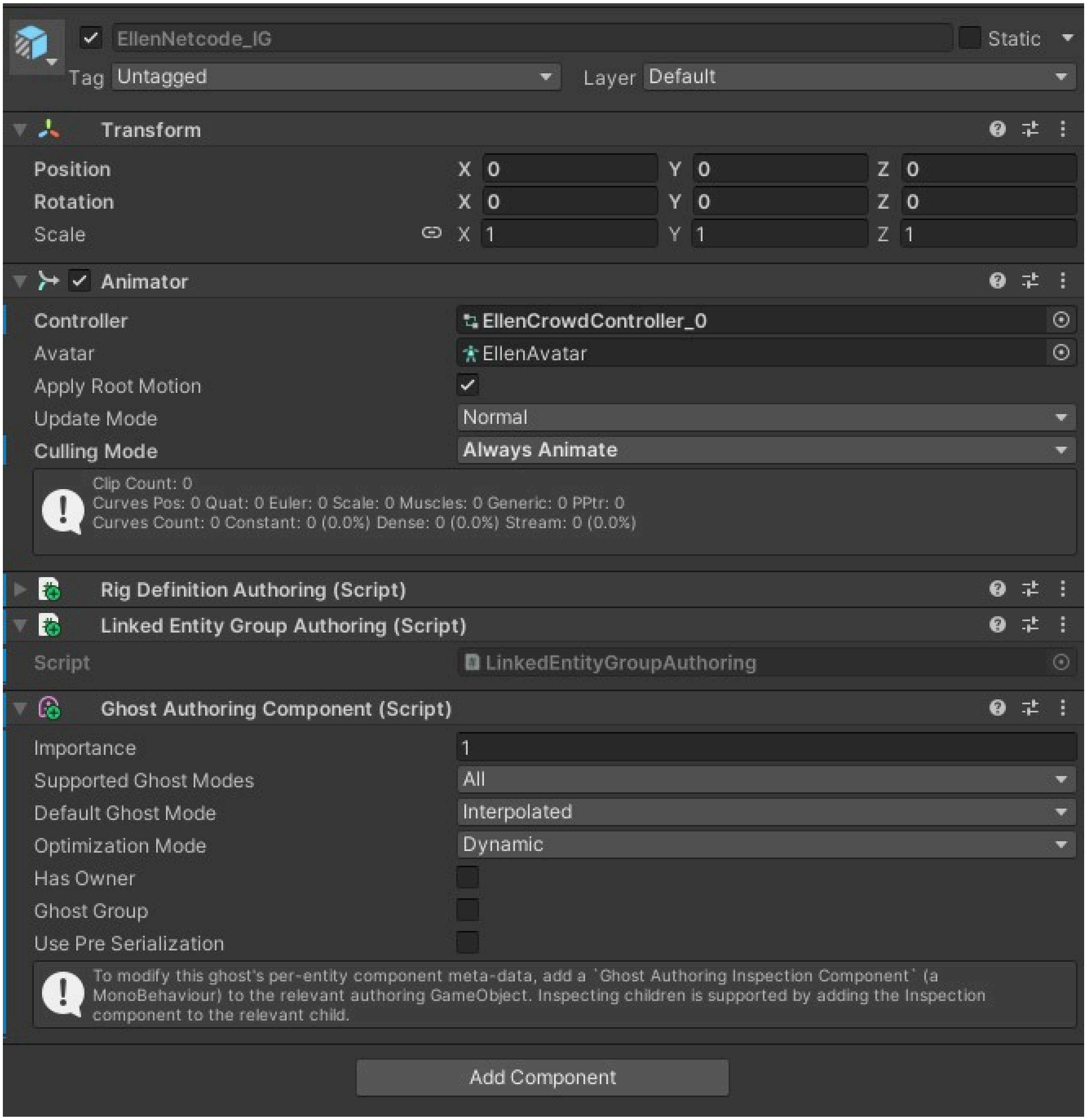
`CPU Animator` writes animated bone positions back to the `LocalTransform` components of corresponding bone entities. No special preparation step is needed to work on bone attachments properly. Just make the attachment entity a child of required bone entity.

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.



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



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

CPU Animator GPU Animator

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.

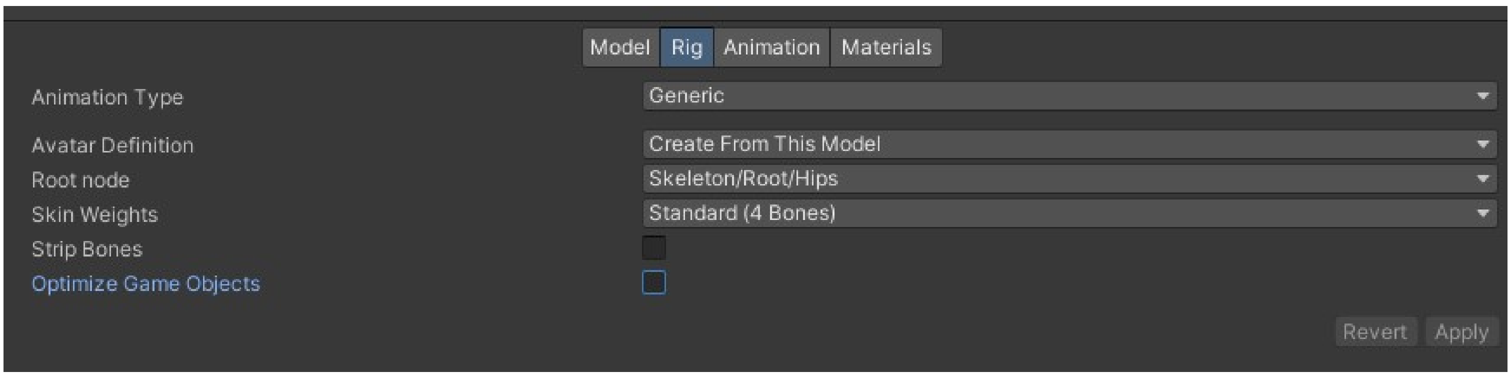
Ragdoll

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.

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:

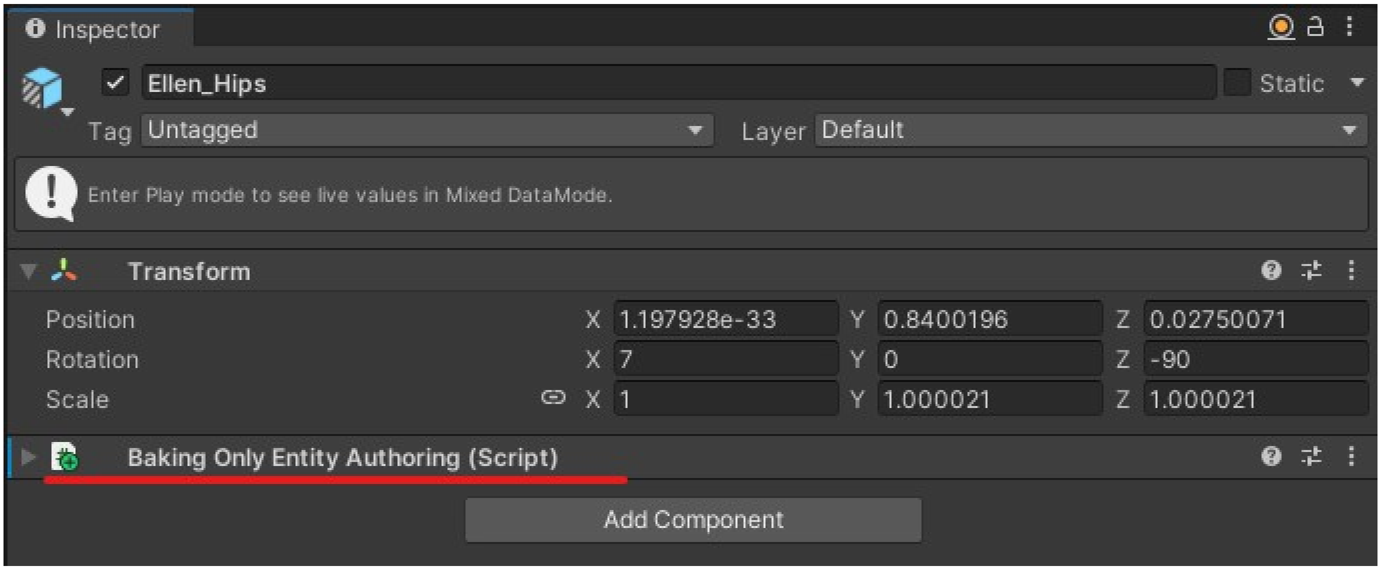


Rukhanka reads full hierarchy information from Unity avatar during the baking phase, so `Optimize Game Objects` checkbox should be **unchecked**.

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

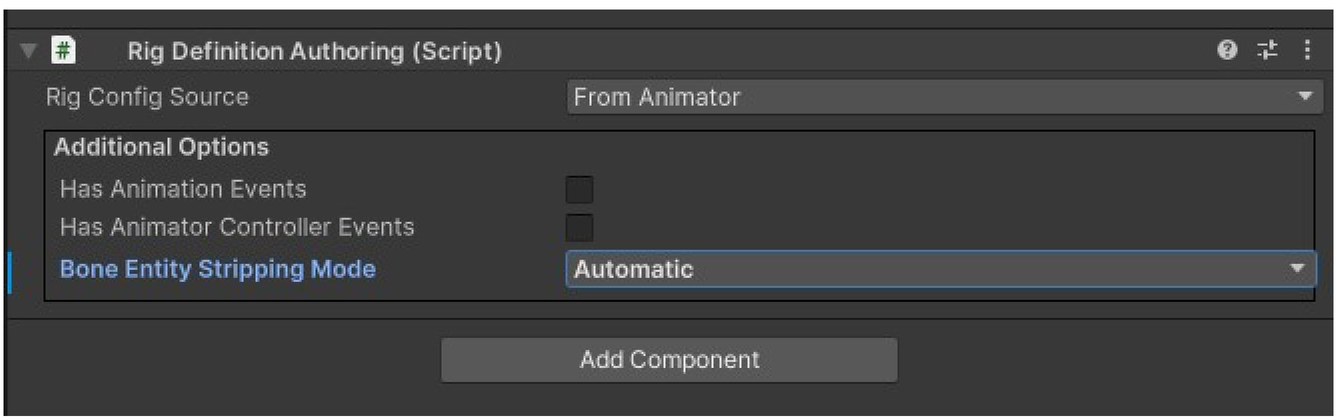
BakingOnlyEntityAuthoring script

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.



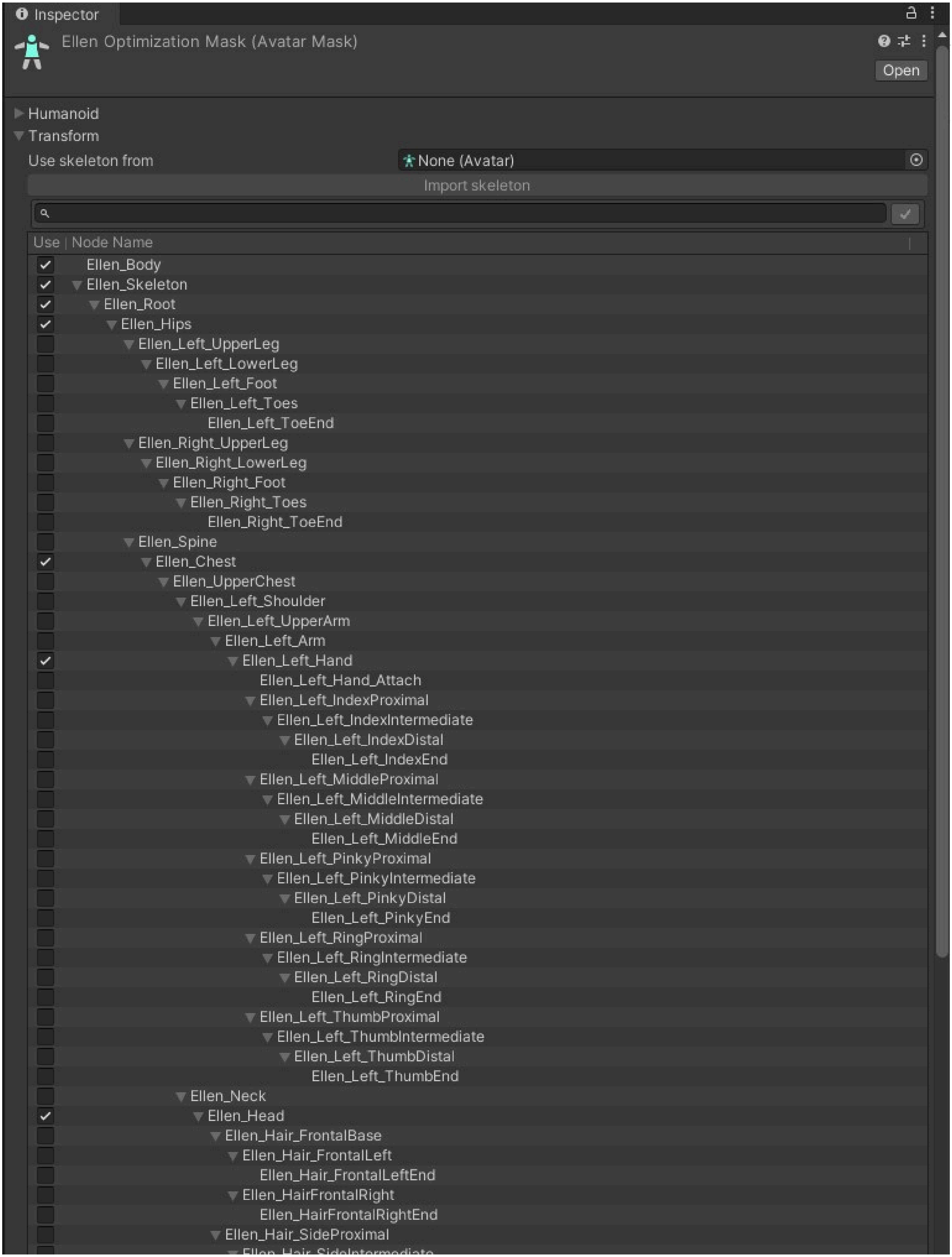
Automatic unreferenced bone entity removal

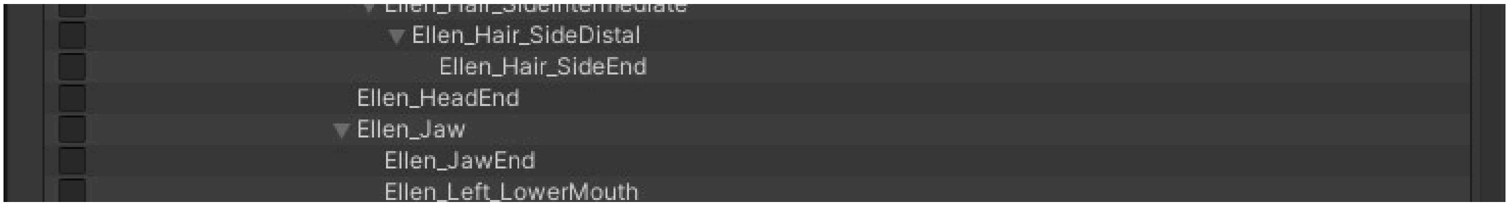
Rukhanka provides an option for automatic bone entity removal of all unreferenced bones. I.e. if no component requires this particular bone entity (for example, physics collider, IK entity, attached object, etc.), then this bone entity will be removed from the world. To activate this mode option `Automatic` should be selected from the `Bone Entity Stripping Mode` selector of the 'RigDefinitionAuthoring' script.



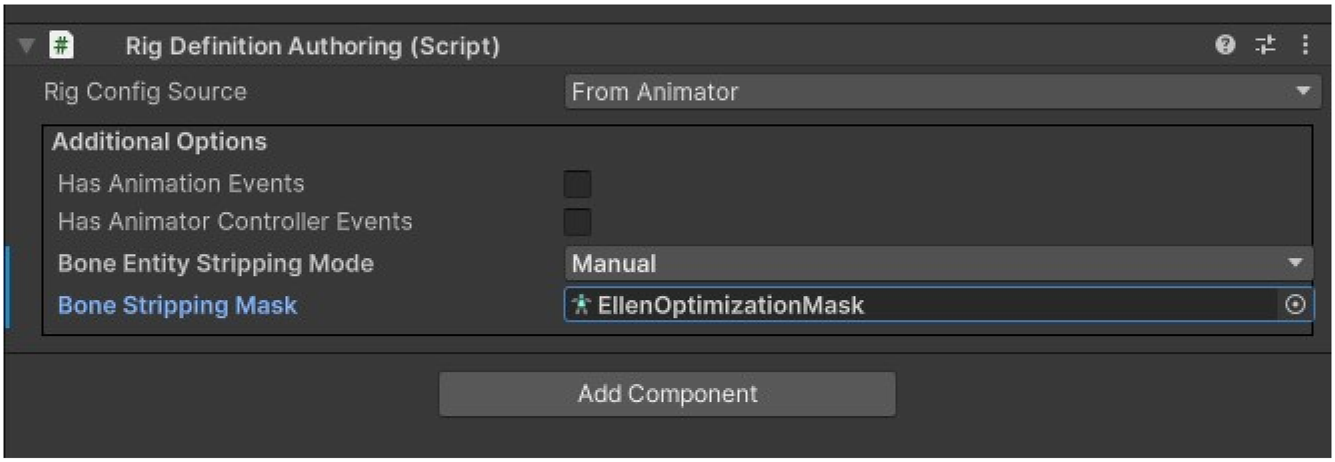
Manual selection of removed bone entities

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:


Inspector
Ellen Optimization Mask (Avatar Mask)
Open
Humanoid
Transform
Use skeleton from
None (Avatar)
Import skeleton
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



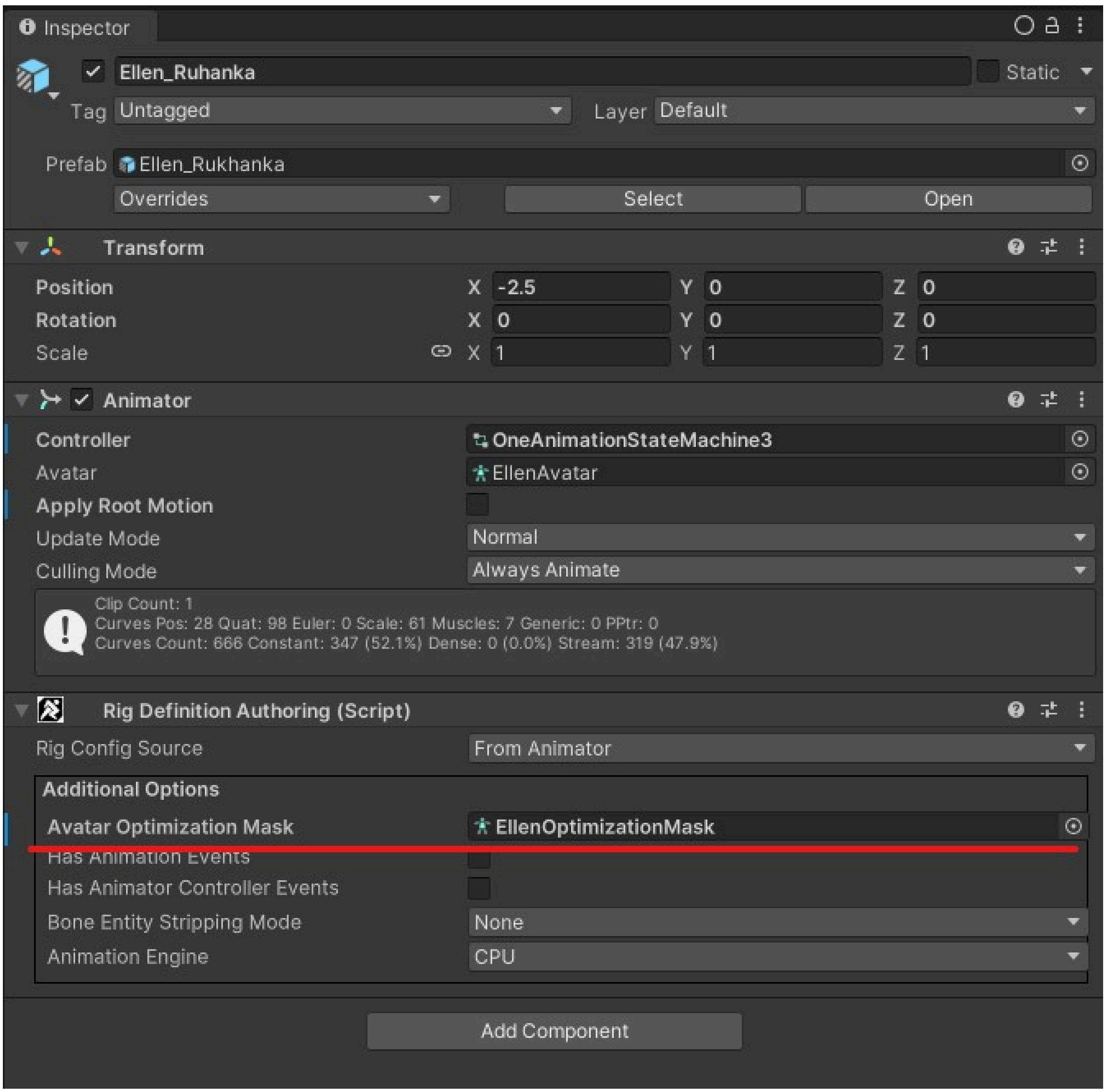
Then set the configured avatar mask in the `RigDefinitionAuthoring` `Bone Stripping Mask` field with `Bone Entity Stripping Mode` set to `Manual`:



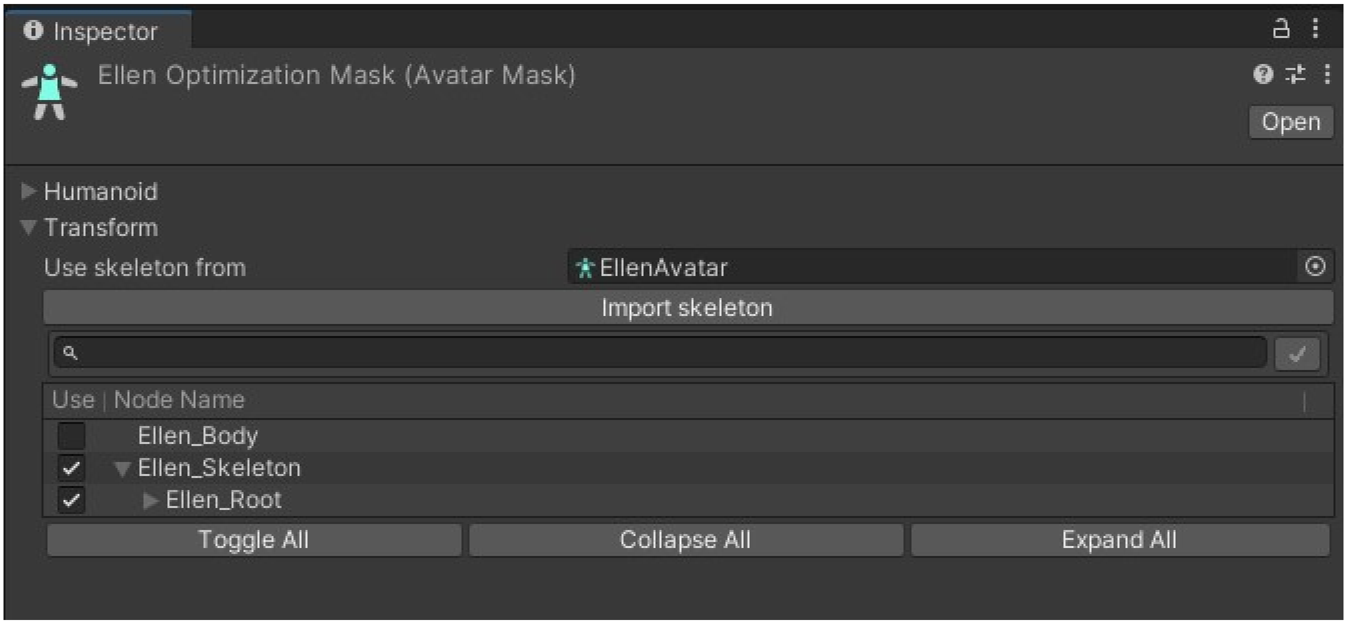
> **IMPORTANT**
>
> By using a bone stripping mask whole entity bone hierarchy will be flattened (i.e. `Parent` component will be removed from bone entities). Unparented bones must have a valid animation track (just one identity keyframe will suffice) associated with it to be animated correctly. Otherwise,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

Optimizing Rig Bone Count

Modular skinned mesh configurations often contain many mesh nodes that are also included in the avatar as separate bones. These bones don't need to be included in the Rukhanka's animation rig, because there are no animation tracks that exist for such bones. Reducing total rig bone count can have a positive impact on overall processing performance. To remove unneeded rig bones one can prepare a special `Avatar Mask` and assign it to the `Avatar Optimization Mask` field of `Rig Definition Authoring` of configured animated object.



The avatar optimization mask is an ordinary `Avatar Mask` created from the animated entity avatar with unneeded bones unchecked.



Animation Frustum Culling

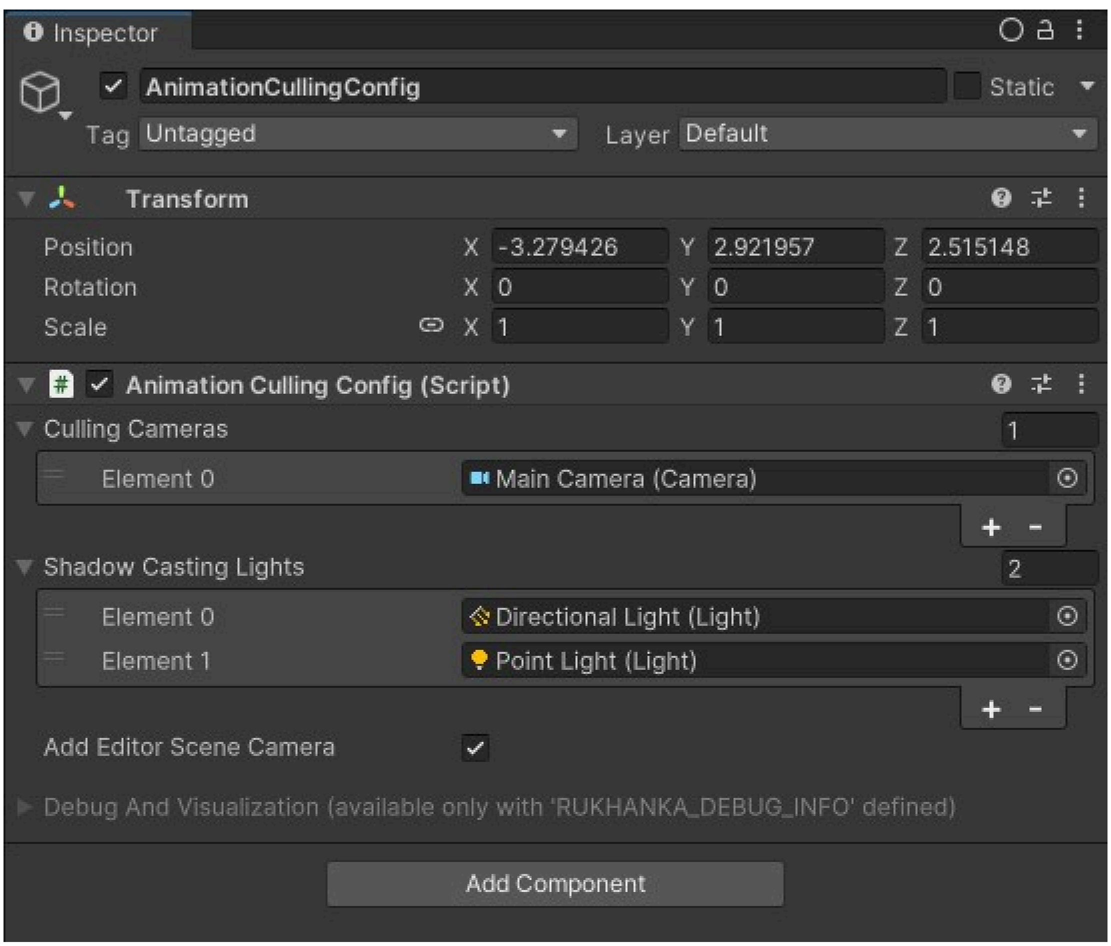
Animation calculation can be disabled for the entities whose renderers are not visible to the camera. **Rukhanka Animation** has advanced capabilities to cull animations with respect to:

- Multiple cameras and their respective frustum volumes to determine object visibility.
- Shadow-casting lights and their respective shadow-casting volumes for renderers shadow visibility determination.

The animation culling configuration consists of two steps: culling environment setup, and per-entity culling preferences configuration.

Animation Culling Environment Setup

Rukhanka must know which relevant scene objects (`Cameras` and `Lights`) it must use for visibility determination. For this purpose, a special script `AnimationCullingConfig` should be used:



This is an ordinary `MonoBehaviour Component` that must reside on the same scene as required culling `Cameras` and `Lights`.

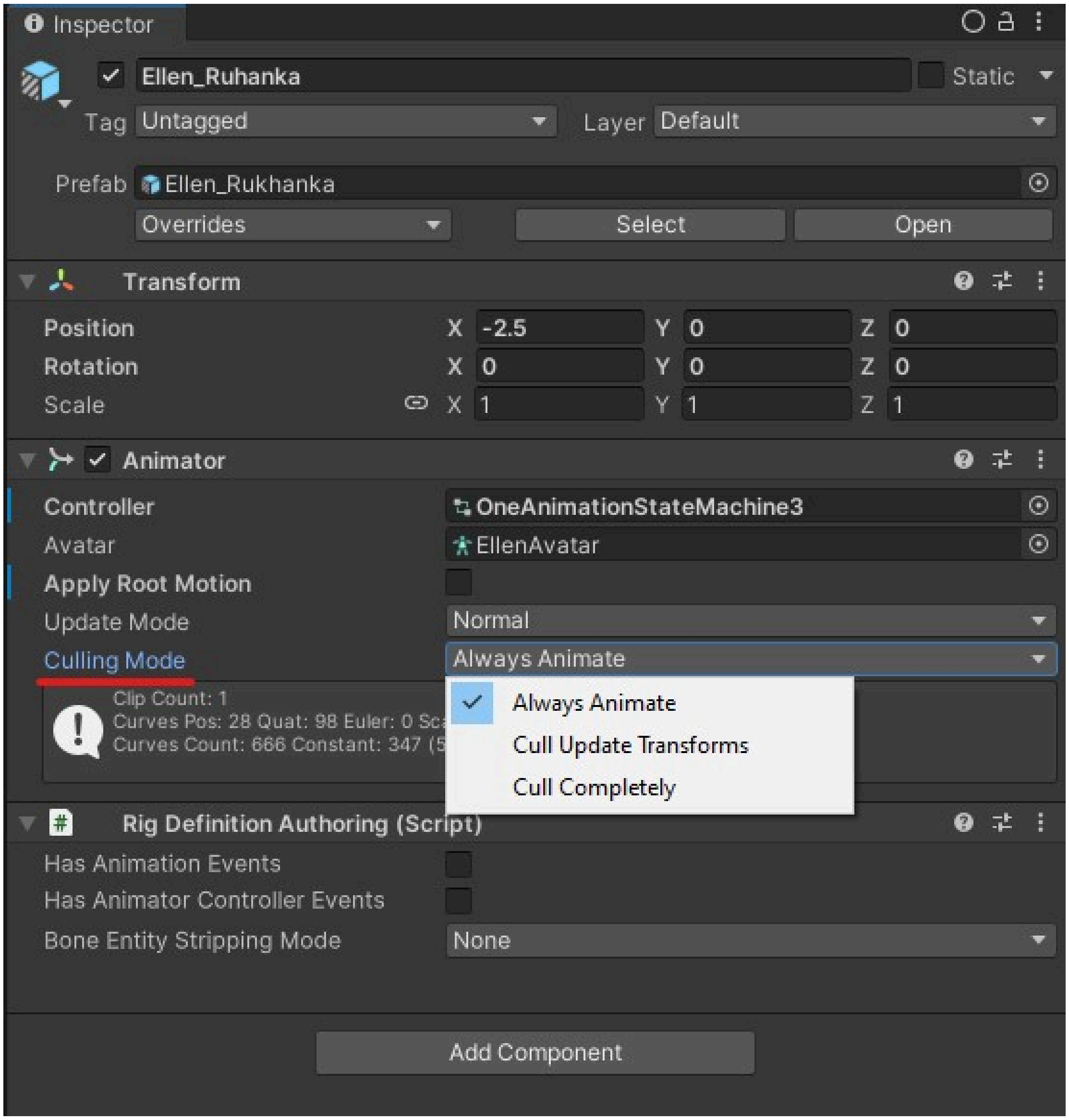
> ⚠ **WARNING**
>
> `AnimationCullingConfig` should not be placed in the entities subscene because it will be destroyed during the baking process

- The **Culling Cameras** array should contain all Unity's `Cameras` used for object visibility determination.
- **Shadow Casting Lights** is an array of shadow-casting `Lights` which shadows should be properly accounted for during object visibility determination.
- The **Add Editor Scene Camera** checkbox is used for adding an editor scene camera to the **Culling Cameras** list.
- The **Debug And Visualization** dropdown contains visualization options for object visibility debugging.

Animated Entity Culling Setup

Animation culling can be configured on a per-object basis.

Unity's `Animator` **Culling Mode** property controls **Rukhanka's** animation culling functionality:



There are two options available:

- **Always Animate** option - do not use animation culling for this entity. Animations will be calculated regardless of visibility.
- **Cull Update Transforms** and **Cull Completely** options have the same meaning - stop animation calculation for entities invisible (and their shadows invisible also) for cameras defined in `AnimationCullingConfig` `Culling Cameras` array.

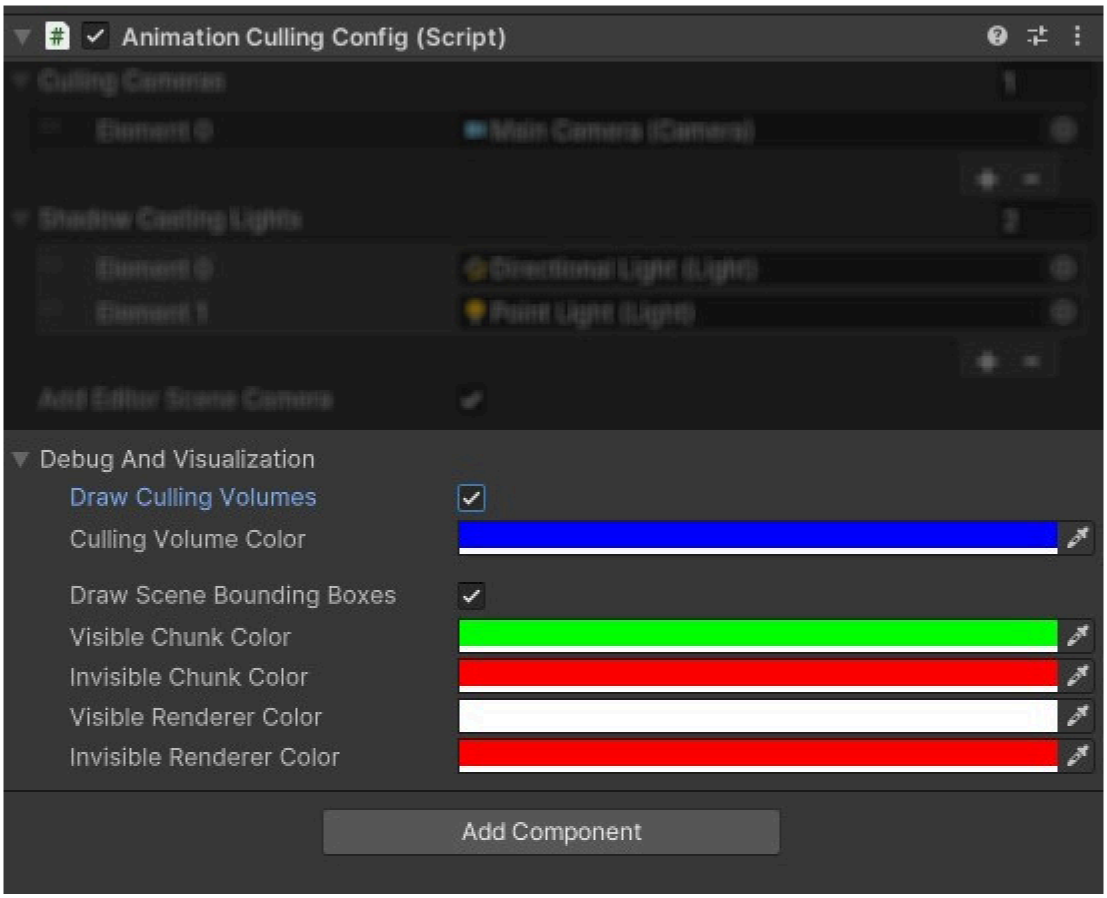
> **INFO**
>
> Note that `Root Motion`, `User Curves`, and `Animation Events` are still processed even for invisible entities.

> **IMPORTANT**

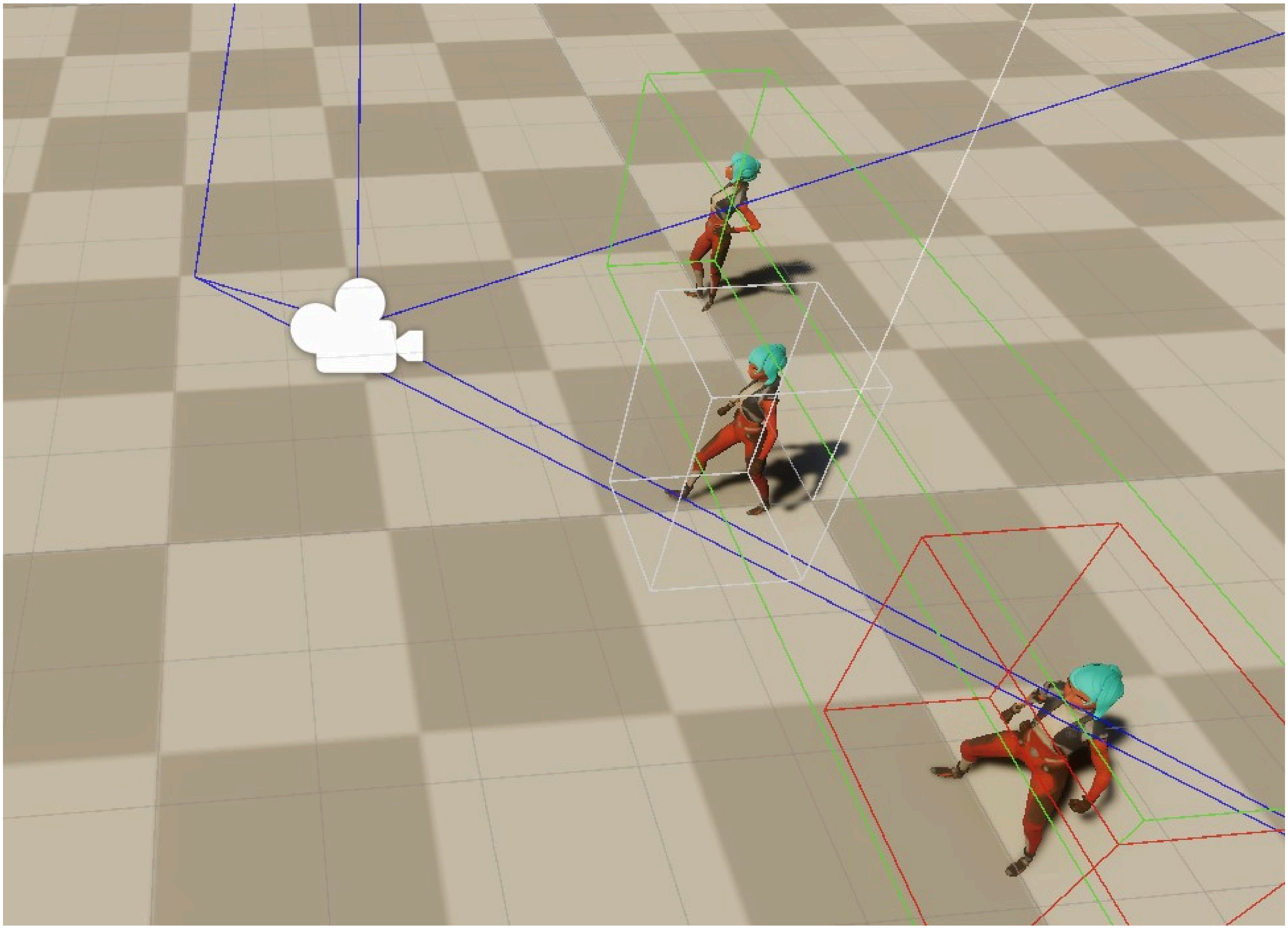
> **Rukhanka** runtime will produce an error if `Animator Culling Mode` is configured to cull animations but `Animation Culling Config` is absent on scene.

Animation Culling Visualization

With `RUKHANKA_DEBUG_INFO` defined `AnimationCullingConfig` additional `Debug And Visualization` options become available.



- With **Draw Culling Volumes** checkbox culling volumes are rendered with **Culling Volume Color**.
- **Draw Scene Bounding Boxes** checkbox controls visualization of chunk and individual renderers bounding boxes with appropriate colors:
  - **Visible Chunk Color** is used for visible chunks bounding box rendering.
  - **Invisible Chunk Color** is used for invisible chunks.
  - **Visible Renderer Color** is used for visible renderers bounding box drawings.
  - **Invisible Renderer Color** is used for invisible renderers.



Half-precision Deformation Data

Skinned vertex stored internally on GPU in the following structure:

```
struct DeformedVertex
{
    float3 deformedVertexPosition;
    float3 deformedNormal;
    float3 deformedTangent;
}
```

Memory required to store this structure is: `(3 + 3 + 3) * sizeof(float) = 36 bytes` of memory for each deformed vertex.

Rukhanka can be switched to use half-precision floating point numbers for deformation data instead of default single-precision ones. In this mode, a deformed vertex will be presented in the form of:

```
struct DeformedVertex
{
    half3 deformedVertexPosition;
    half3 deformedNormal;
    half3 deformedTangent;
}
```

This structure requires only `(3 + 3 + 3) * sizeof(half) = 18 bytes` of memory.

This mode has the following characteristics:

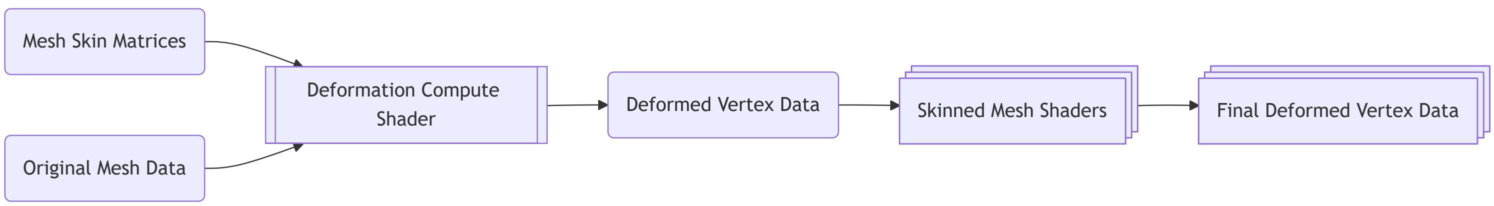
- The memory bandwidth required to transfer deformation data during skinned mesh computation and application is reduced by half. Because this factor greatly influences skinning performance, reducing bandwidth can help to increase of efficiency of mesh skinning.
- Memory storage requirements for deformed vertex storage are reduced by half.
- Deformation data precision is reduced, and, potentially, could introduce animation artifacts. Make sure to carefully test your animation set for potential visual discrepancies after enabling this mode.

Rukhanka Animation can be switched to this mode by using Settings Dialog.

In-place Skinning

Preskinning Pass

The default skinning process can be described using the following diagram:

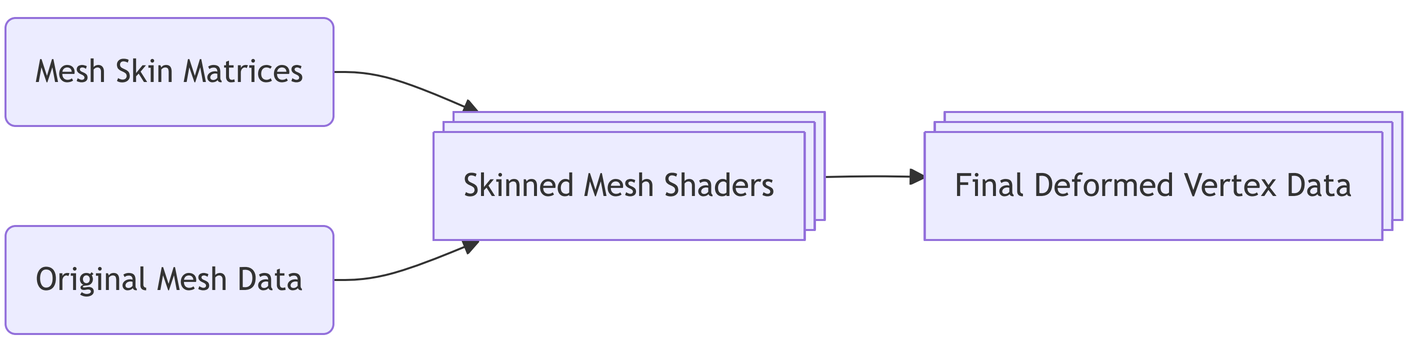


The depicted pipeline has the following properties:

- The deformation compute shader is executed only once per frame regardless of the number of actual render passes.
- Skinning vertex shader uses already deformed data (made by deformation compute shader) as the current frame vertex position, normal, and tangent.
- The memory storage required to hold frame deformation data can be significant and can be calculated using the following formula: `Number of deformed frame vertices * size of the deformed vertex (36 bytes for single-precision data, and 18 for half-precision data)`.
- Skinning performance is mostly limited by memory bandwidth.

Direct Skinning in Vertex Shader

Rukhanka Animation can be switched to the `In-place skinning` operation mode. This mode can be enabled in **Rukhanka's** Settings Dialog. In-place skinning process can be shown using the following diagram:



The in-place skinning pipeline has the following properties:

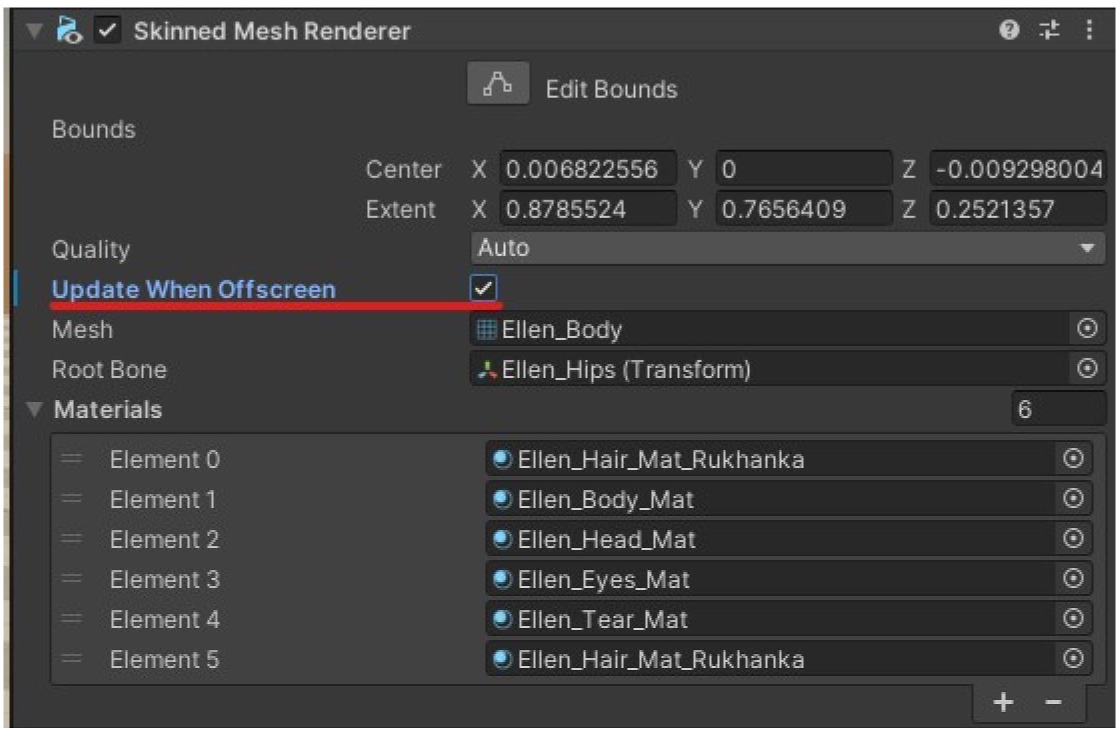
- There is no one-frame deformation computation step.
- Skinning deformation is performed directly in mesh vertex shaders. The deformation code is executed every time the mesh vertex shader is executed. This means that every mesh rendering pass will perform its own skinned mesh deformation computation (applying skin matrices to the mesh vertices). For renderer pipelines, that have many frame render passes (depth pre-pass, shadow map cascade passes, etc.), this can introduce significant algorithmic pressure on the vertex shader stage.
- Memory bandwidth is not an issue for this approach.
- Memory requirements are significantly lower because there is no intermediate deformed vertex data storage buffer.

> **TIP**
>
> The performance of the skinning part of the deformation pipeline can depend on many factors (used renderer pipeline, number of render passes, etc). To pick the best skinning mode, it is advisable to do performance tests with your project.

Renderer Bounding Box Recalculation

CPU Animator   GPU Animator

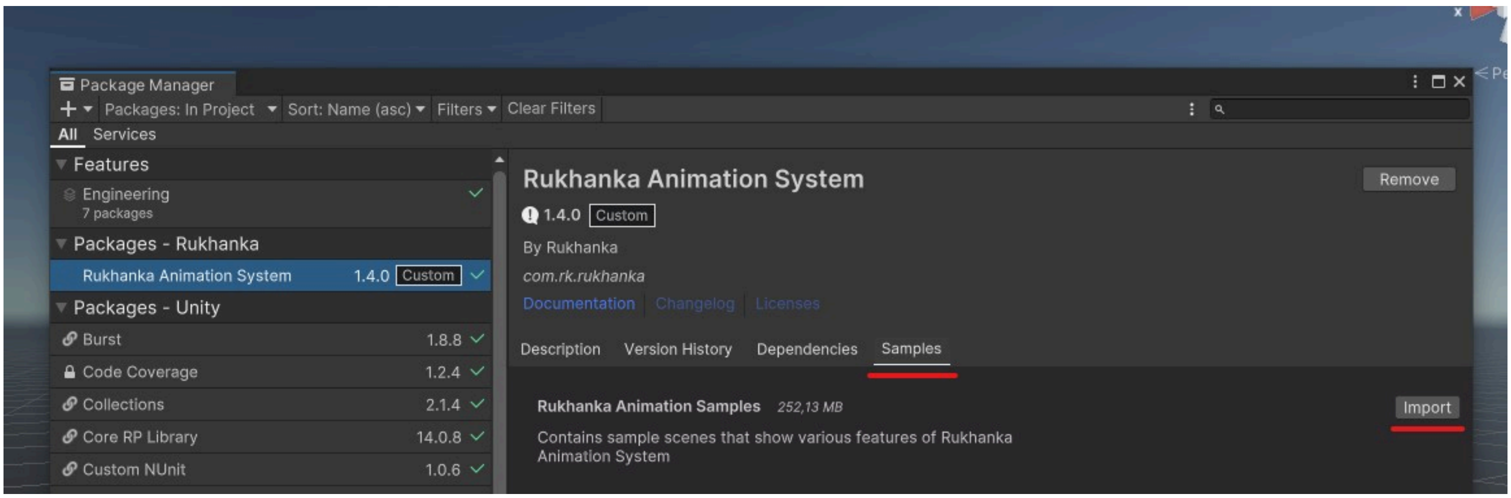
Rukhanka can recalculate animated skinned mesh renderers bounding box from skeleton bone poses. To enable this for a particular `SkinnedMeshRenderer` just enable the `Update When Offscreen` checkbox in Unity's `SkinnedMeshRenderer` configuration inspector:



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.

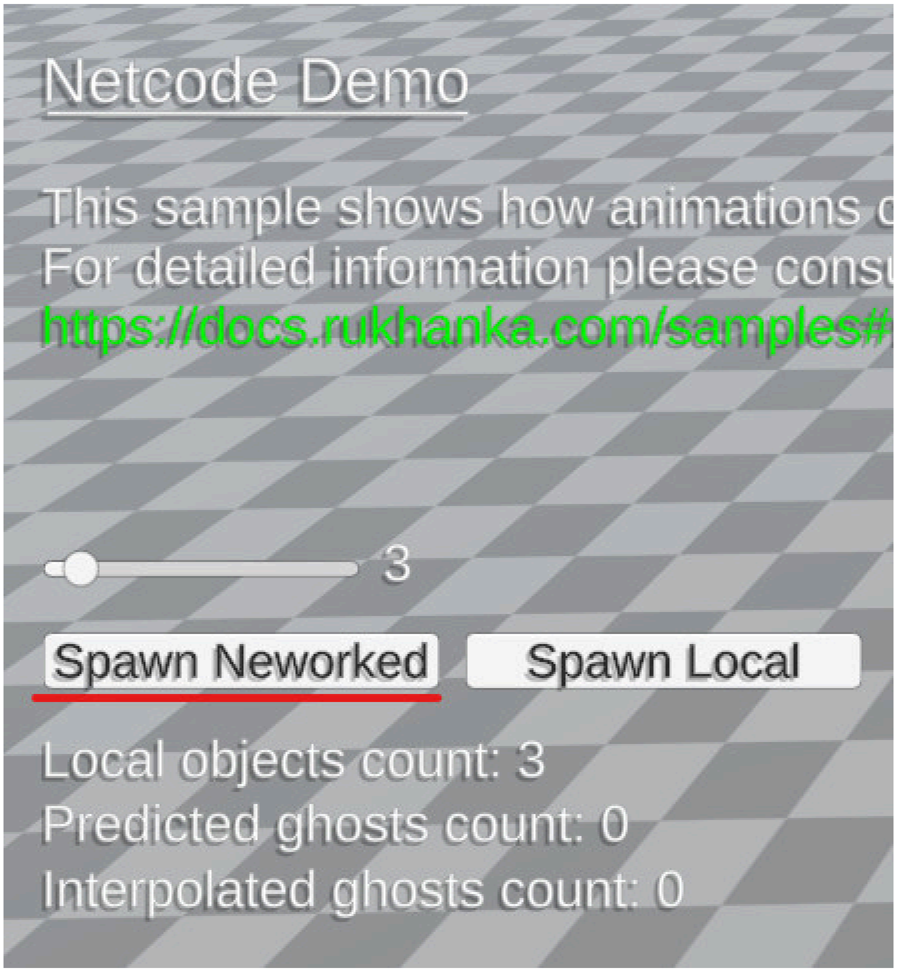
Three types of objects that can be instantiated on the scene:

- 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:

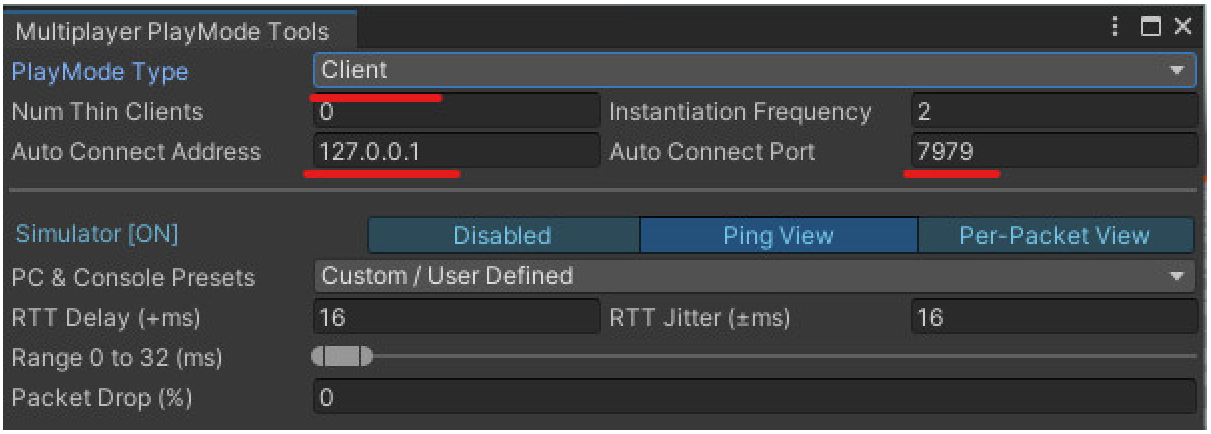
- Predicted ghosts. They are colored with green-tinted materials:

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



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:



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.

Animation and Animator Events

This sample shows a usage example of animation and animator controller events - the new feature of v1.5.0. A simple system reacts to animation events and spawns indicator particles. In reaction to controller events particle color will be changed.

Inverse Kinematics

Various IK algorithms, that are available in **Rukhanka** distribution, are presented in this sample.

Animation Culling

This sample shows **Rukhanka's** ability to skip animation processing for invisible entities. Skinned mesh renderer bounding box recalculation is also presented in this demo.

Scripted Controller

Rukhanka can work without the `Mecanim` animator controller. In such cases, manual animation submission from code is required. This sample shows manual animation submission and animation parameters manipulation.

Blend Shapes

Rukhanka can animate arbitrary floating point values. If the animated value is blend shape weight for a particular skinned mesh renderer, then this value will be propagated to the corresponding `Entities.Graphics` `BlendShapeWeight` component. This sample shows this functionality.

Runtime Skinned Mesh Replacement

You can attach new skinned meshes to the animated rig in runtime. Because of the relative complexity of skinned mesh entity representation (one skinned mesh entity and one or many skinned mesh render entities, one for each submesh), extra work is required. This sample introduces the methods of how to add skinned mesh to the existing animated rig. Note that those skinned meshes come from different separate prefabs. The core of the swapping method is located in the `ModularRigInitPartsSystem.cs` file.

Procedural Animation

A sample showcasing how animations can be generated dynamically at runtime using code and simulation rather than pre-made keyframes. It demonstrates techniques like blending, physics-driven motion, and parameter-based control to create responsive, adaptive character movement that reacts naturally to gameplay.

Character Controller

This sample shows relatively complex animator controller and its work in cooperation with the Unity Character Controller package.

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 hash 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 from settings dialog.

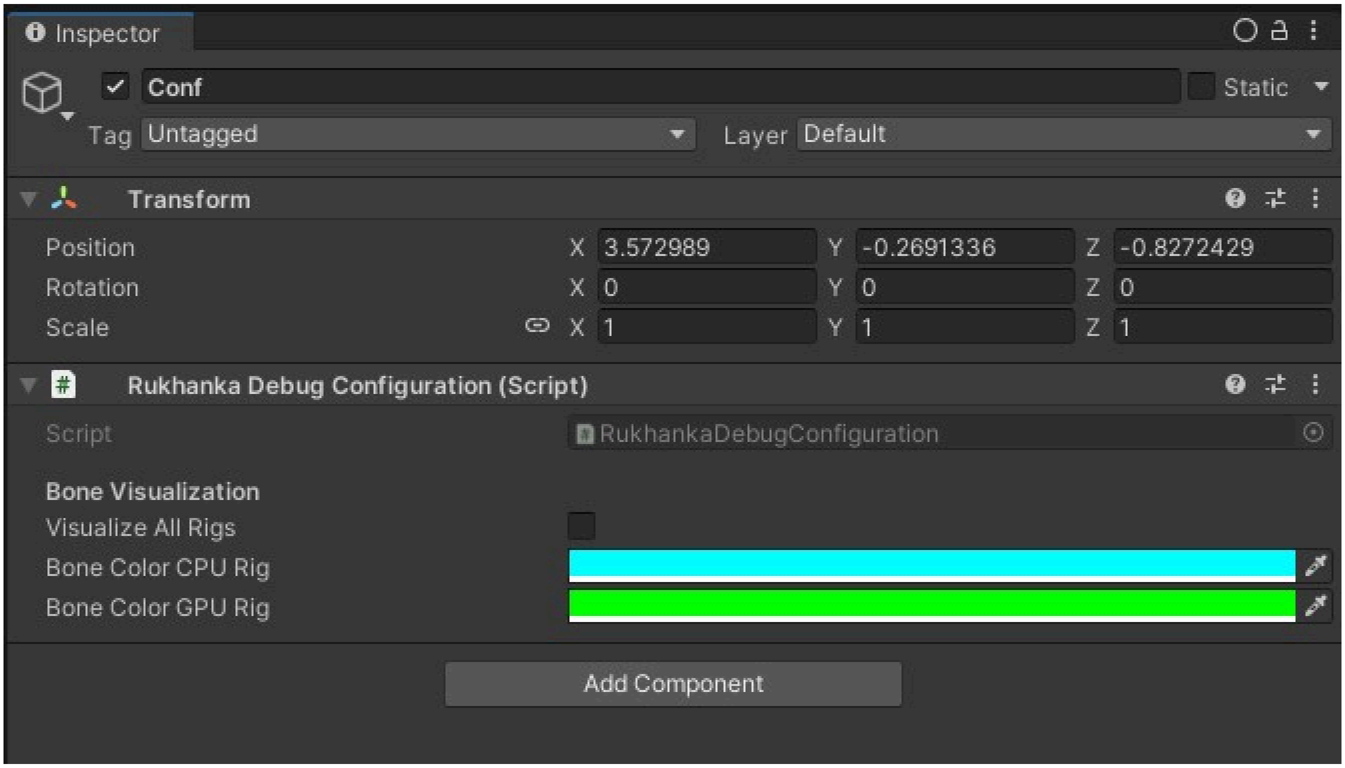
With validation mode enabled, **Rukhanka** will add to all internal structures its corresponding string fields (FixedString or BlobString for `Burst` compatibility where appropriate). Watching these members in the debugger and logging makes it much easier to investigate and fix problems in animations.

Animation resource names available with validation layer greatly improve working experience with Blob Inspector and Animation Wayback Machine.

Bone Visualization

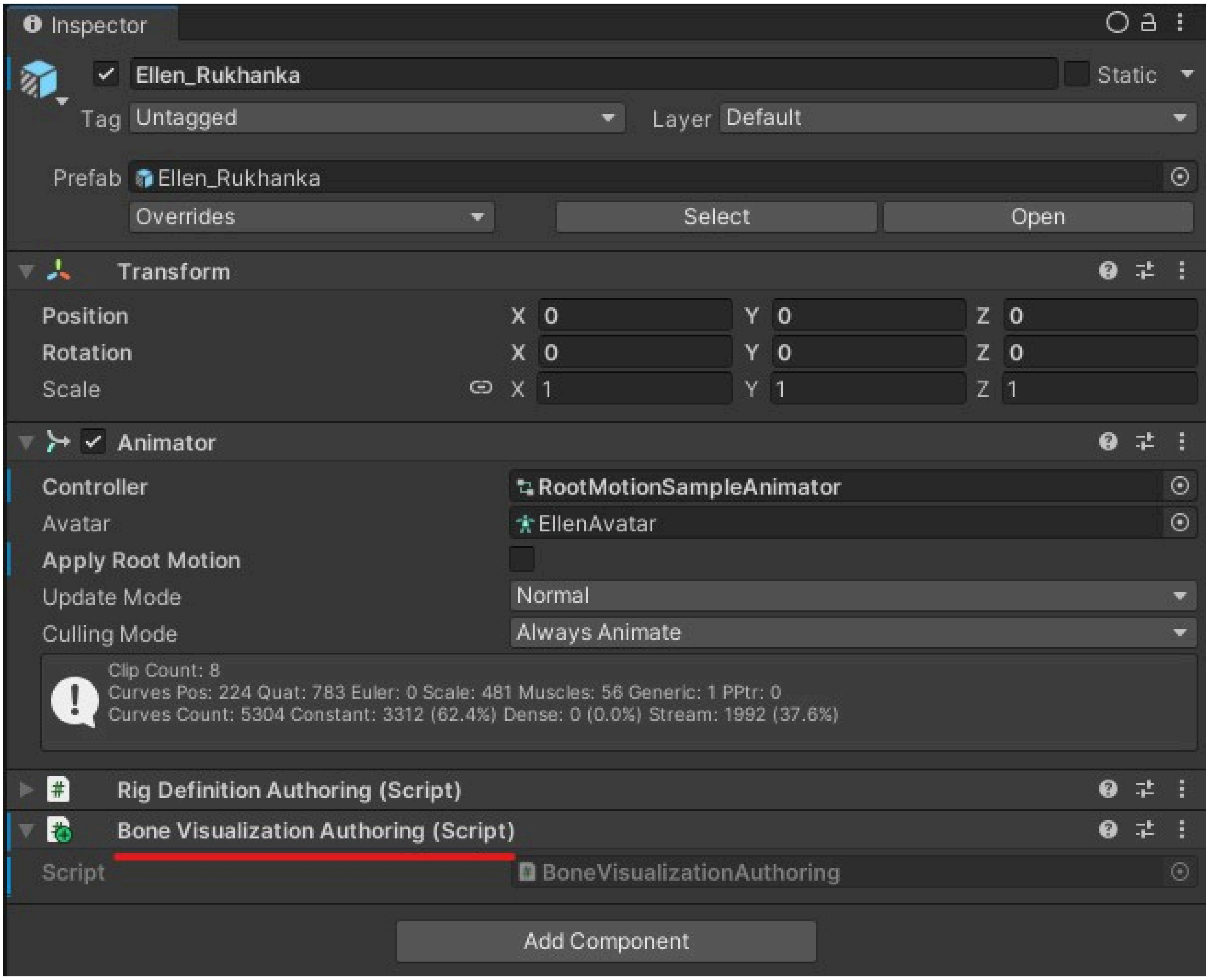
There are two options to enable *Bone Visualization* capability for **Rukhanka** `Rig`:

- Enable bone visualization for all meshes in the scene. Add the `Rukhanka Debug Configuration` authoring component to any `GameObject` inside `Entities Subscene`, and set `Visualize all Rigs` checkbox in it:



- Add the `Bone Visualization Authoring` component to the required animated object.

> ⓘ **INFO**
>
> Bone rendering functionality is completely stripped out without `RUKHANKA_DEBUG_INFO` defined.

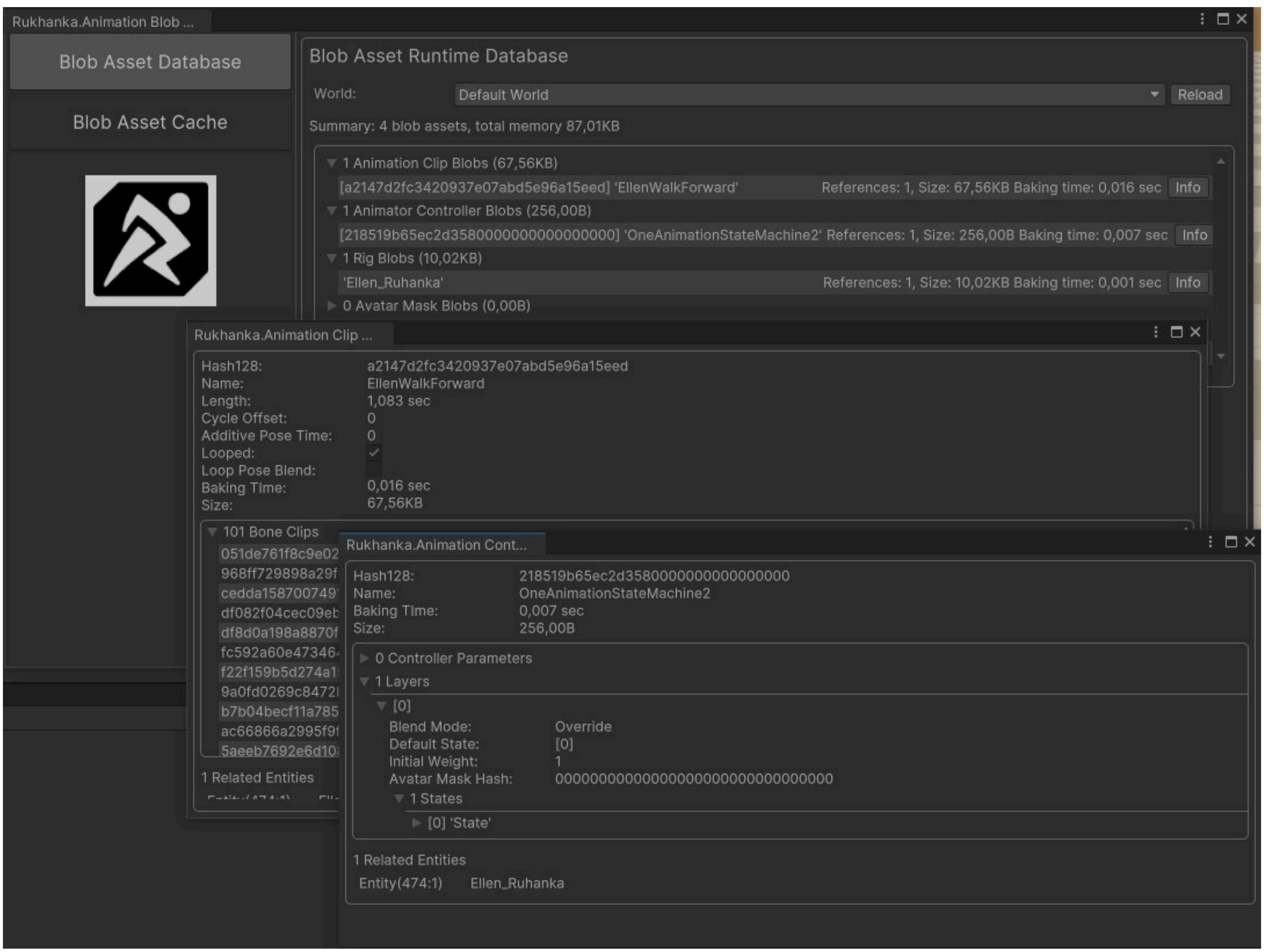

Inspector
Ellen_Rukhanka
Static
Tag Untagged
Layer Default
Prefab Ellen_Rukhanka
Overrides
Select
Open
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Animator
Controller RootMotionSampleAnimator
Avatar EllenAvatar
Apply Root Motion
Update Mode Normal
Culling Mode Always Animate
Clip Count: 8
Curves Pos: 224 Quat: 783 Euler: 0 Scale: 481 Muscles: 56 Generic: 1 PPtr: 0
Curves Count: 5304 Constant: 3312 (62.4%) Dense: 0 (0.0%) Stream: 1992 (37.6%)
Rig Definition Authoring (Script)
Bone Visualization Authoring (Script)
Script BoneVisualizationAuthoring
Add Component

Blob Inspector Dialog

Runtime Blob Information

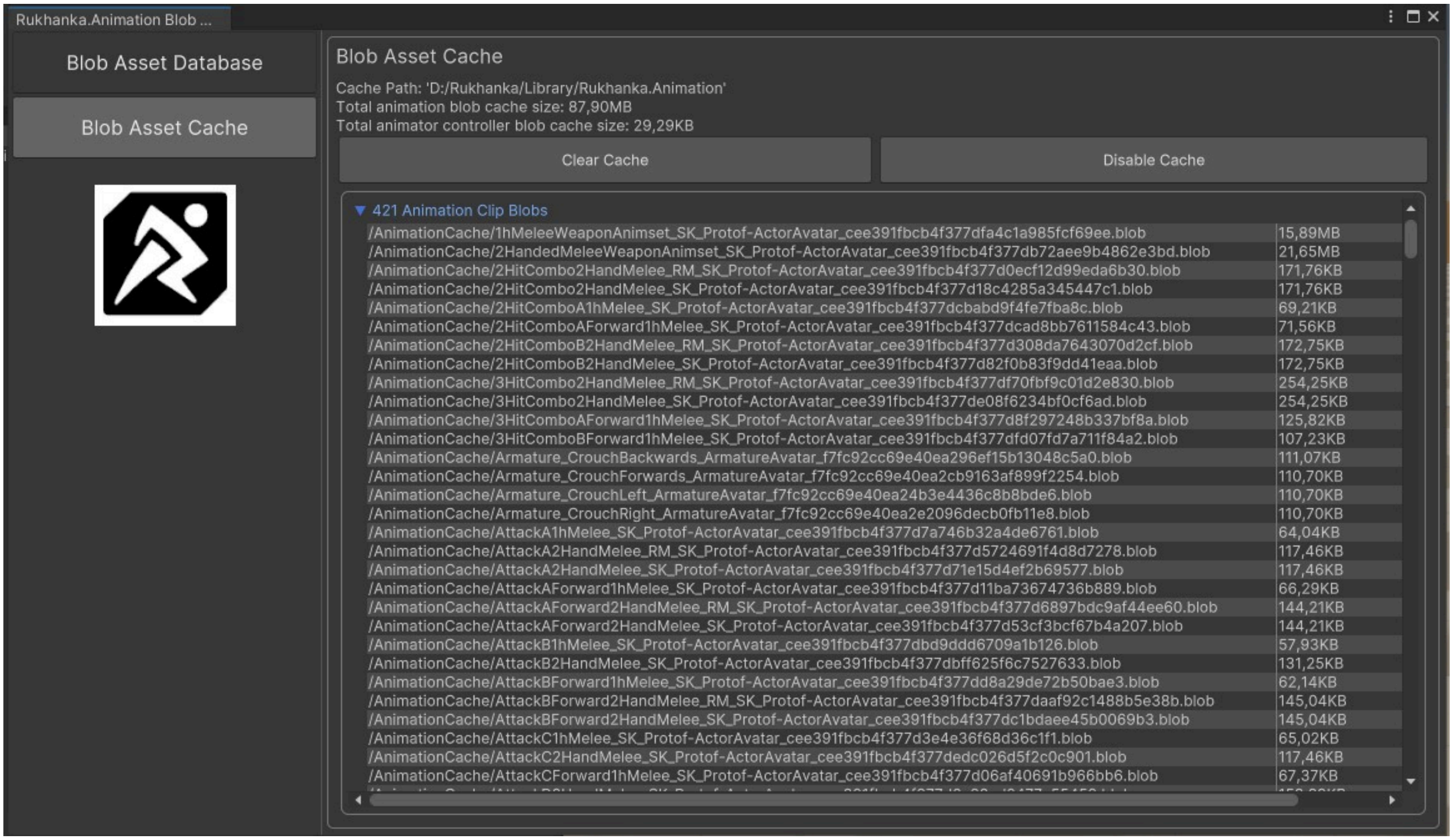
All baked blob assets can be inspected with the `Blob Inspector` editor window. It can be found under `Window->Rukhanka.Animation` Unity Editor menu. It contains detailed information about baked animation clips, animator controllers, avatar masks, rigs, and skinned meshes. The `Blob Asset Database` pane contains all runtime blob information for the selected world.

> **WARNING**
>
> Name and baking time fields will be only available in Debug and Validation mode



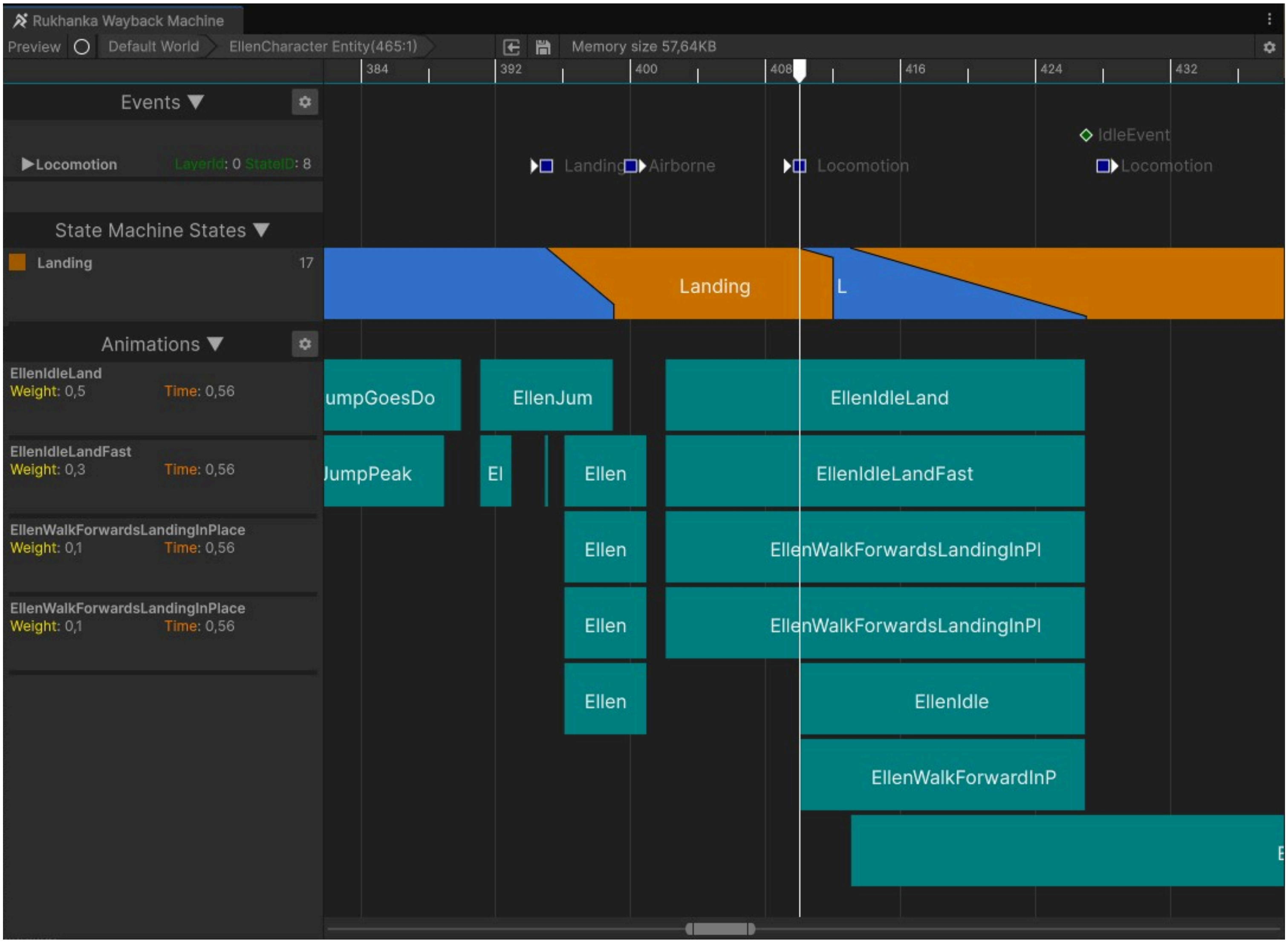
Blob Asset Cache

Rukhanka caches converted animation clip and animator controller blobs to disk. This will significantly speedup iteration times with open subscenes.



Animation Wayback Machine

The `Animation Wayback Machine` is the powerful tool for recording, analyzing, and replaying animation sequences, controller states, and animation events. The `Wayback Machine` window can be opened using `Window->Rukhanka.Animation->Animation Wayback Machine` option in the Unity Editor menu.



> ⚠ **WARNING**
>
> Asset and resource names are only available in Debug and Validation mode

Recording Animation Sequence

Select the ECS world and the required animated entity using the toolbar controls of the `Wayback Machine` window, and press the `Record` button to begin the recording process. `Rukhanka Animation` records the following animation data using a fixed-step sampling method:

- Played animations
- Animator controller states
- Animator controller transitions
- Animation events
- Animator controller state events

Replaying Animation Sequence

After selecting the target entity using the same toolbar controls and pressing the `Preview` button, `Rukhanka Animation` will overrides the ordinary animation workflow with the data read from recorded information. Applied animation data snapshot is taken from timeline marker. Timeline marker detailed information is shown in left pane of the `Animation Wayback Machine` window.

Saving and Loading Recorded Data

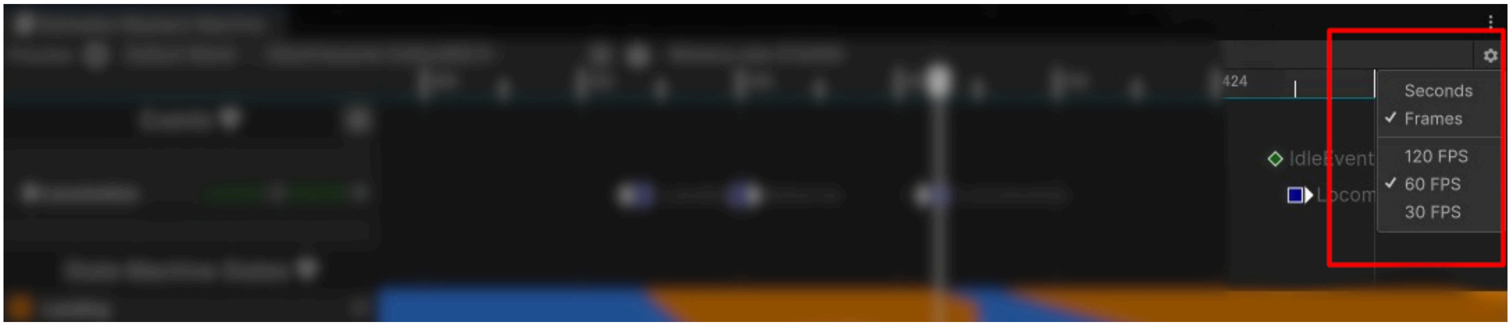
Recorded animation data can be saved to file using toolbar `Save` button for later analysis.

Configuration

There are several options that can be configured:

Animation sample rate

The sample rate defines how often `Rukhanka Animation` samples animation data during recording.

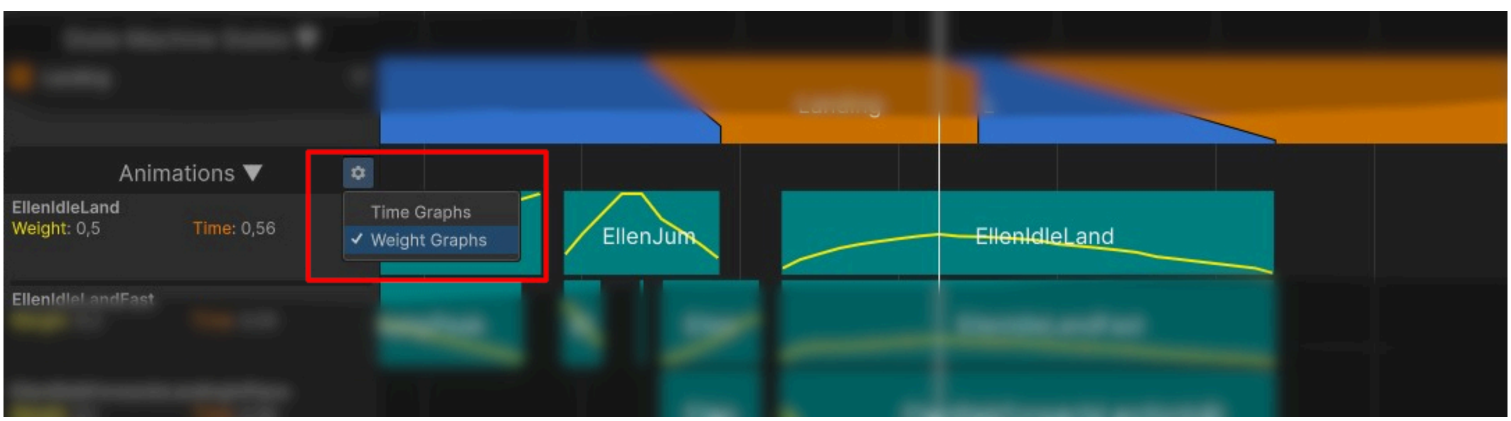


Animation Time Graphs

Time graph curves can be rendered on top of the recorded animation bars.

Animation Weight Graphs

Weight graph curves can be rendered on top of the recorded animation bars.



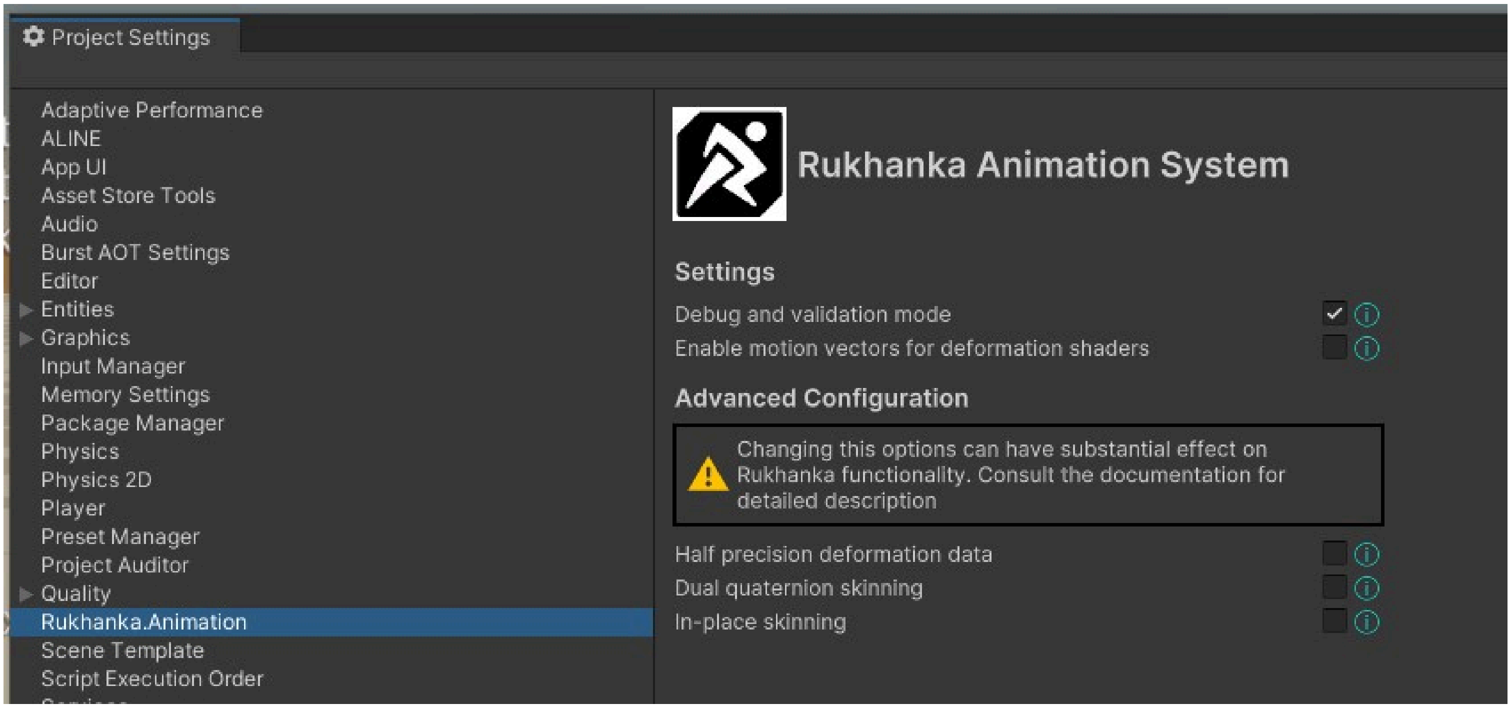
Event Labels

Animation and animator state events labels can be turned on or off.

384
392
400
408
416
424
432
Events
Event Labels
IdleEvent
comotion
LayerId: 0 StateID: 8
Landing
Airborne
Locomotion
Locomotion

Settings Dialog

Rukhanka's functionality can be configured with a settings dialog that can be found under the "Edit->Project Settings->Rukhanka.Animation" menu.



Settings

Debug and validation mode

This option controls enabling/disabling the Extended Validation Mode of **Rukhanka** operation.

Enable motion vectors for deformation shaders

With this option, motion vector calculation can be enabled/disabled for **Rukhanka** deformation shaders.

Advanced Configuration

Half-precision deformation data

Enabling this option will enable half-precision deformation data storage project-wide.

Dual quaternion skinning

Dual quaternion skinning is the volume-preserving skinning method. The conventional linear blend skinning (LBS) approach tends to volume loss on joints. Dual quaternion skinning (DQS) is an alternative to LBS, intended to avoid loss of volume problem.

> **INFO**
>
> Dual quaternion skinning has a bit higher computation costs than linear blend skinning.

In-place skinning

Enabling this option will enable in-place skinning project-wide.

Changelog

[2.7.2] - 09.01.2026

Fixed

- Improper CPU animation code disabling when GPU animated entities were used exclusively.
- `RigDefinitionComponent` lookup was not properly passed into `UpdateSkinnedMeshBoundsJob`.

[2.7.1] - 04.01.2026

Fixed

- LOD components were added incorrectly during prefab baking.
- Non-uniform scale was handled incorrectly for meshes that act as GPU attachments.

[2.7.0] - 01.01.2026

Added

- Skinned mesh renderers can be split into multiple independent skinned meshes, one per source material. Use the `SkinnedMeshSplitOnSubmeshEntitiesAuthoring` component on the authoring skinned mesh renderer to enable this functionality.

Changed

- The `ComputeFrameSkinnedMeshWorkloadSizesJob` job was made multithreaded, leading to better thread utilization and improved performance.
- Internal structures were simplified by replacing some hash maps with entity components. This simplifies the code and provides small performance improvements.
- Animator override animation final blobs were previously created each frame. They are now cached.

- The `GetInstanceId` function has been replaced with `GetEntityId` for `Unity 6000.4+` due to former's deprecation.
- The minimum supported versions of the Entities and Entities.Graphics packages are now v1.4.3 and v1.4.16, respectively.

Fixed

- A null reference exception during incorrect `HumanBodyPartOverrideAuthoring` handling in the authoring code.
- Obsolete code that was accidentally left in when the `RUKHANKA_ENABLE_DEFORMATION_MOTION_VECTORS` script symbol was defined.
- An incorrectly declared `SkinMatrix` GPU buffer type. This fixes potential skinned mesh corruption (particularly on integrated AMD GPUs with older drivers).

[2.6.0] - 08.12.2025

Added

- Human bone type can be overridden by using `HumanBodyPartOverrideAuthoring` on authoring skeleton bone `GameObjects`.
- `RootMotionVelocityComponent` now includes frame delta position and delta rotation fields.

Changed

- The minimum supported versions of the `Entities` and `Entities.Graphics` packages are now v1.2.4.
- Support for the `Entities.Graphics` deformation system has been dropped. The `RUKHANKA_NO_DEFORMATION_SYSTEM` script symbol no longer exists.
- Skinned mesh baking has been reworked. No additional render entities are created for each authoring skinned mesh. This change closely resembles `Entities.Graphics` mesh render baking with the `ENABLE_MESH_RENDERER_SUBMESH_DATA_SHARING` script symbol defined. The new approach greatly simplifies skinned mesh entity handling and its material-override manipulation code.
- `AnimatedSkinnedMeshComponent` has been renamed to `SkinnedMeshRendererComponent` to better reflect its purpose.

Fixed

- Incorrect handling of transition interruptions in the `ScriptedAnimator.CrossFade` function.
- `GenericDropdownMenu.Dropdown` API deprecation warning in Unity 6000.3+.
- Incorrect scale application in GPU bone-attachment handling code.

[2.5.2] - 26.11.2025

Added

- Support for skinned meshes with multiple duplicated bone names. This allows proper animation of skinned meshes created with various mesh-merge tools (particularly the `Synty® Sidekick Character Creator` tool).

Changed

- In-place skinning is now enabled by default.

Fixed

- Animation culling was being ignored during blend shape computation.
- Fixed incorrect motion ID computation for looped blend tree cases.

[2.5.1] - 09.11.2025

Added

- The `Crowd` scene has `Despawn` button now.

Fixed

- A 1D blend tree parameter with a value greater than the defined maximum caused incorrect mesh scaling.
- Animation events were emitted incorrectly during rapid controller state changes.

- The `Animation Wayback Machine` could record duplicate events or skip animation events due to differences between the sampling and simulation system rates.
- Fixed incorrect hierarchy traversal in dynamic bone chains during baking.

[2.5.0] - 01.11.2025

Added

- Animation Wayback Machine.
- Third person character controller sample.

Changed

- `AnimatorParameterAspect` and `AnimatorStateQueryAspect` no longer implement `IAspect` interface due to its deprecation. Refer to the migration process for further details.

Fixed

- The blend tree computation code no longer outputs zero-weighted animations.
- Corrected motion submission in 1D blend tree for blend values greater than blend tree's maximum.
- Resolved code generation errors when using `RUKHANKA_WITH_NETCODE`.

[2.4.0] - 05.10.2025

Added

- Animator controller transition interruption support.
- Animator controller layer sync feature.

[2.3.0] - 07.09.2025

Added

- Dynamic bone chain procedural animation algorithm.
- Procedural animations sample scene.
- Animation events support for GPU-animated entities.
- Animation culling support for orthographic camera type.

Fixed

- `AnimationStream.SetWorldPosition` API not working incorrectly.
- Incorrect calculation of layer weights when an animation state had an empty motion.
- Animation events emitted incorrectly for animations introduced in the current processing frame.
- `Amplify Shader Editor` deformation node worked incorrectly.
- Error in GPU animator Euler-to-quaternion rotation conversion code.
- Memory leak during simultaneous rig-to-skinned mesh remap table creation.
- Incorrect rendering of skinned mesh without defined root bone.
- Properly handle disabled GPU-animated entities.
- Incorrect "Enable Motion Vectors" help URL in project settings.
- "Attempt to draw with missing binding" warning and missed GPU attachments in Edit mode.
- Incorrect handling of negative motion time for non-looped animations.
- Incorrect handling of negative motion time in root motion processing code.
- Incorrect layer weight calculation in user curve processing code.

Changed

- GPU attachments can now be standalone baked objects.
- Compute shaders are now DXC compiler–compatible, fixing shader compilation errors on Xbox.
- Reworked "single deformation compute dispatch" approach: compute shader now uses a constant `numthreads` attribute, improving compatibility with mobile platforms.
- Removal dependency on `ENABLE_DOTS_DEFORMATION_MOTION_VECTORS` script compilation symbol.

[2.2.1] - 03.05.2025

Added

- Weight dependent tip node rotation for `TwoBoneIK` algorithm.

Fixed

- Incorrect thumb finger humanoid bone name.
- Missing blend shape data buffer bindings in shader for in-place skinning.
- Incorrect handling of non-uniform scaling in skinning shaders.
- Out-of-bounds graphics buffer access in the skinning shader, which caused rendering errors on some Android platforms with Vulkan.
- Switched skinned mesh bone weights buffer update scheme from the `SetData` API to `SparseUploader`. The former caused incorrect data clear during partial uploads on MacOS and missing meshes as result.

[2.2.0] - 06.04.2025

Added

- Configuration settings dialog.
- New skinning shader subgraphs for `Unity Shader Graph`, `Amplify Shader Editor`, and `Better Shaders`.
- Dual quaternion skinning support.
- Half-precision deformation data.
- In-place skinning.
- Multi-submesh GPU attachment meshes support.
- New `RootMotionVelocityComponent` for custom root motion handling. New configurable root motion mode in `RigDefinitionAuthoring`.

Changed

- Deprecation message about default `Entities.Graphics` deformation shader usage. Its support will be removed in next **Rukhanka** release.

Fixed

- The rig optimization mask had incorrect bone order.
- Properly ignore missing animation tracks during animation weight calculation.
- Internal new skinned mesh hash map capacity was increased to prevent "Hash map is full" exceptions during big number different meshes instantiation.
- Missed `AvatarMask` baking code in `AnimationAssetSetBaker`.
- Incorrectly working `AnimationStream.SetWorldRotation` API.
- Update skinned mesh render bounds code worked incorrectly.

[2.1.0] - 26.02.2025

Added

- `SkinnedMeshRenderEntity` component buffer. This buffer holds references to all baked render entities for particular skinned mesh. This buffer helps simplify material overrides and runtime mesh addition/removal manipulation.
- Rig bone count optimization mask.
- "Better Shaders" shader building tool node for making deformation-compatible shaders.
- Runtime animator layer speed ability.
- Runtime skinned mesh replacement sample.

Changed

- Improved Blob Inspector dialog UI.
- `Debug.Drawer` HDRP/URP shaders merged into one for both SRPs. This removes shader compilation warnings about incompatible SRP.

Fixed

- Incorrect behavior of GPU animation engine when animated entity is disabled (via `Disabled` component).
- The deformation system wrong update order which led to one frame missing mesh for renderers with the configured LOD group.
- Incorrect keyframe index check for two-frame tracks in GPU animator.

- Deformed meshes with blend shapes are not properly counted during the internal buffers resize operation of the deformation system. This could lead to the "Hash map is full" runtime exceptions.
- The middle joint drift of the `TwoBoneIK` algorithm if no animation is playing for the rig.
- Incorrect absolute entity pose computation of IK targets that are non-root entities.
- Incorrect humanoid bone mapping for toe bones.

[2.0.0] - 25.12.2024

Added

- Innovative GPU animation engine. Full-featured animation processing routines implemented on GPU using compute shaders.
- `ScriptedAnimator.CrossFade` API which mimics Unity's `Animator.CrossFade`.
- The additive reference pose frame was always taken from the current animation. A separate pose frame was needed for advanced blending scenarios. This ability was added to the animation clip blob and baker.

Changed

- The avatar mask representation was changed to bitmask. This lowers memory usage and speeds bone masking during animation processing.
- Animation blob asset structure refactoring. Tracks are stored in a more uniform way now. This leads to a simpler and performant sampling code.
- Perfect hash table reworked. Lookups are much more performant now.
- `MaterialOverrides` propagation from authoring skinned mesh entity to all sub-mesh entities.
- Additive and loop pose calculations require first and/or last animation frame sampling. Previously this was done by sampling with keyframe search (generic approach). Now, special direct first and last frame sampling functions were added, speeding up the entire computation loop.

Fixed

- Incorrect blob asset registration that comes from multiple scenes.
- Incorrect additive animation layer weight calculation.

- Properly handle `Disabled` animated entities and skinned mesh entities.
- Preventing infinite values in animator controller state times.
- Headless and unsupported configurations improperly disable GPU-related systems.
- `Entities.Graphics` deformation system treats deformed mesh index 0 as an invalid index. Valid indices started from 1. **Rukhanka's** deformation system uses 0 based index, which breaks motion vector functionality for the first deformed entity in the world.
- Memory leak during blend shape copying in the deformation system.
- Blend shape incorrect vertex count calculation during GPU upload.

[1.9.2] - 04.11.2024

Fixed

- Improperly handled `Disabled` entities.
- Improper handling of controller zero length states.

[1.9.1] - 09.10.2024

Added

- Documentation for `ScriptedAnimator.PlayAnimatorState` function.
- `GetStateIndexInControllerLayer` function that can be used to find the state ID for `ScriptedAnimator.PlayAnimatorState` API.

Changed

- Making `CopyBuffer` and `ClearBuffer` compute kernels with smaller thread group size (128) to be able to work on low level-mobile GPUs. Large data sizes are handled via multiple dispatches.
- The `Skinning` compute kernel now has variations with different workgroup sizes. The correct one is selected depending on hardware capabilities in runtime.

Fixed

- Skinned mesh to rig remap table cache can run out of capacity.

- Incorrect additive layers weight calculation.
- Incorrect blend shape deformation when both skinning and blend shapes are used together.
- Inverse kinematic targets positions were calculated from entity `LocalTransform` components. This leads to one frame position lag for targets that are children of animated bones. Now the actual position is taken from the animation stream in this case.
- Indexing animation layers were wrong, if some layers had zero weights.

[1.9.0] - 03.09.2024

Added

- Rukhanka Deformation (skinning) System.
- `ScriptedAnimator` animator state playback API.

Changed

- Several fixes and improvements were made to the `TwoBoneIK` algorithm.
- Skinned mesh blob inspector window with additional data.

Fixed

- Blob assets loaded from the cache were incorrectly registered in the blob database, leading to duplication of blob assets and memory leaks.
- Incorrect Euler to quaternion conversion rotation order.
- Mistakenly removed manual bone stripping mask processing.
- `EntityCommandBuffer` obsolete `EntityQueryCaptureMode.AtRecord` usage.

[1.8.1] - 31.07.2024

Added

- `ResetTrigger` API for `AnimatorParametersAspect`.

Fixed

- Corrupted animation blob data after scene reload.
- Unknown symbol `FixedStringName` in AnimatorStateQuery documentation page
- Baked animation events have an uninitialized `nameHash` member variable.

[1.8.0] - 24.06.2024

Added

- Blend shape support.
- Blend shape sample scene.
- `ScriptedAnimator` blend tree (1D, and all types of 2D) API.

Fixed

- The `AmplifyShaderEditor` deformation node is now disabled correctly during authoring `GameObject` rendering.
- Blob cache invalid file characters handling.
- Animation baker persistent memory leaks.

[1.7.1] - 07.06.2024

Fixed

- Fixed compilation errors during standalone builds creation.

[1.7.0] - 06.06.2024

Fixed

- Individual trigger reset for each animator controller layer.

Added

- Ability to work without baked `Animator`.
- Baking code rewritten from scratch. With a cleaner and more straightforward approach baking code becomes faster and uses less memory.
- Blob Inspector dialog to review baked blob assets.
- Blob Cache. Opened subscenes baking times are significantly improved.
- `Animator Override Controller` animations now can be switched in runtime.
- `Avatar Mask` can be toggled in runtime.
- `Scripted Animator` sample scene.

Changed

- `Avatar Mask` sample scene with runtime mask toggling control.
- `Animator Override Controller` sample scene with runtime animation toggling control.

[1.6.4] - 17.05.2024

Fixed

- 'LockBufferForWrite: Multiple uploads in flight for buffer' error.
- Incorrect error message and assert in `EmitAnimationEventsJob`.
- Error in `AnimationStream.Dispose` function in case of invalid stream rig data.

Added

- Ability to completely compile out `DebugDrawer` via `RUKHANKA_NO_DEBUG_DRAWER` script symbol.

[1.6.3] - 16.03.2024

Changed

- Reverting `Entities` dependency to version `1.0.16` with appropriate changes. No functionality was affected.

[1.6.2] - 14.03.2024

Fixed

- Animation events edge cases (start and end animation).
- Correcting events for looped animations.
- Exception during renderers bounding box recalculation if skinned mesh renderer root bone property is null.

[1.6.1] - 9.03.2024

Fixed

- `DebugDrawer` incorrect line renderer in Unity versions newer than 2022.3.17f1.
- Authoring object copy for animation sampling is not created if avatar is missing on animator.

[1.6.0] - 7.03.2024

Fixed

- User curves incorrect multiple layers blending.
- `IK` targets incorrect world pose calculation if the animated entity is not a hierarchy root.

Added

- Animation frustum culling ability.
- Skinned mesh renderer bounding box recalculation ability.
- Sample showcase scene to demonstrate animation culling and bounding box recalculation features.
- `IsInTransition` utility function of `AnimatorStateQueryAspect`.

Changed

- Updated `Entities` dependency to version `1.2.0-pre.6`
- Making `DebugDrawer` a client only system.

[1.5.1] - 10.02.2024

Fixed

- Removed empty `DebugDrawer` draw calls in case of no primitive submitted.
- Server world missed the animation injection system group.
- Some memory leaks during baking.

Added

- Duplicated bone names checker during baking.

Changed

- `AnimationStream` bone hierarchy recalculation rework. There is no need for `RebuildOutdatedBonePoses` explicit calls anymore. All dependent bones will be recalculated automatically during `Get` calls. `AnimationStream` is derived from an `IDisposable` interface now, and disposal is required after its usage.

[1.5.0] - 01.02.2024

Fixed

- The `BoneTransform.Inverse` function worked incorrectly with respect to scale.
- User curves were parsed incorrectly in humanoid animation clips.

Added

- Animation modification injection point represented by `RukhankaAnimationInjectionSystemGroup`.

- `AnimationStream` structure to simplify working with animation data.
- Several inverse kinematics algorithms:
  - Aim.
  - Override transform.
  - Forward And Backward Reaching Inverse Kinematics (FABRIK).
  - Two Bone.
- Animation events.
- Animator controller events.
- The debug bone renderer was extended to be able to draw various primitive geometry (lines, triangles, cones, cubes, etc.) and moved to separate `DebugDrawer` assembly. It is independent from other **Rukhanka** assemblies and can be used as a standalone library.
- Two new samples: Events and IK.

Changed

- The package's internal name has been changed to `com.rukhanka.animation`.
- Bone stripping functionality has an additional `Automatic` mode now.
- The `RukhankaDebugConfiguration` new options to log all animation and animator controller events.
- Multiple bone visualization systems can run. This is useful to display skeletons in multiple worlds (particularly when using NetCode).

[1.4.2] - 15.12.2023

Fixed

- The last syncpoint of `AnimationProcessSystem` job has been removed by moving all frame initialization code into a separate job.
- Fixed incorrect excessive internal hash map capacity setting.
- Fixed memory leaks during blob creation in several baking systems.
- Internal perfect hash tables gain better validation. This change fixes rare creation errors from valid data.
- Correct handling of inexistent conditions during state machine baking.
- Fixed enter transition->exit transition sequence that appeared in one frame.

- Fixed warning of implicit usage of `ToNativeArray()` call.
- The transition offset were measured in normalized time. It is correctly in seconds now.

Added

- Sub-state machine transitions was implemented.

Changed

- Animator parameter usage documentation has been changed to emphasize `AnimatorParametersAspect` as the preferred runtime interface to parameters data.

[1.4.1] - 06.10.2023

Changed

- 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.

Fixed

- Incorrect root bone indexing for humanoid rigs.
- Incorrect non-root motion of humanoid rig hips.

[1.4.0] - 28.09.2023

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] - 11.08.2023

Fixed

- Fixed compilation errors during standalone builds creation.

[1.3.0] - 10.08.2023

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] - 20.06.2023

Fixed

- Fixed compilation errors during standalone builds creation.

[1.2.0] - 14.06.2023

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] - 30.05.2023

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] - 06.05.2023

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] - 28.03.2023

Fixed

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

[1.0.1] - 19.02.2023

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.

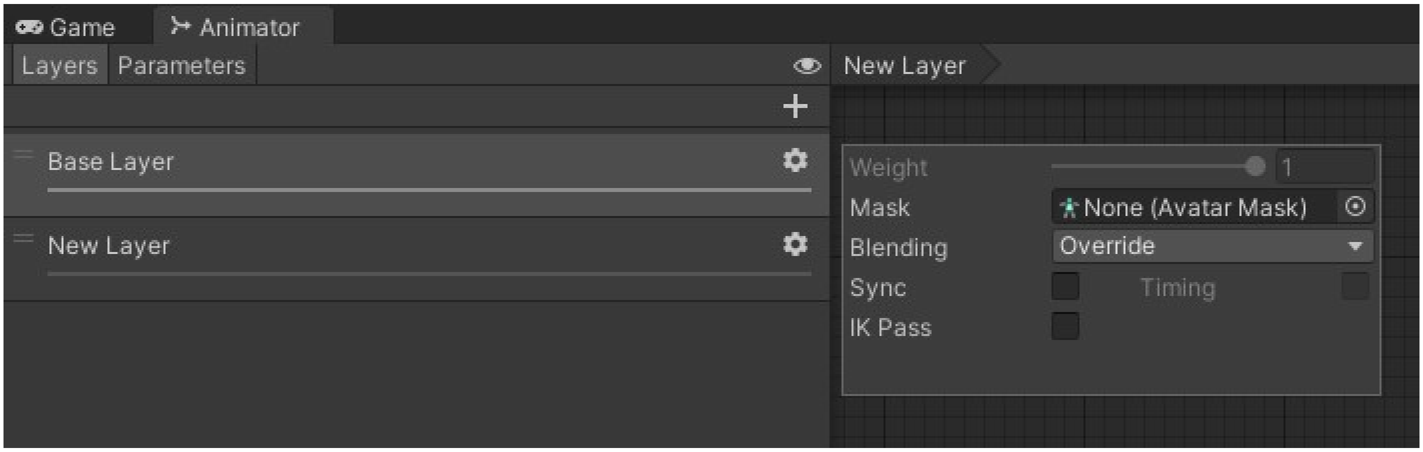
Fixed

- Incorrect handling handling of uniform scale in animations.

[1.0.0] - 10.02.2023

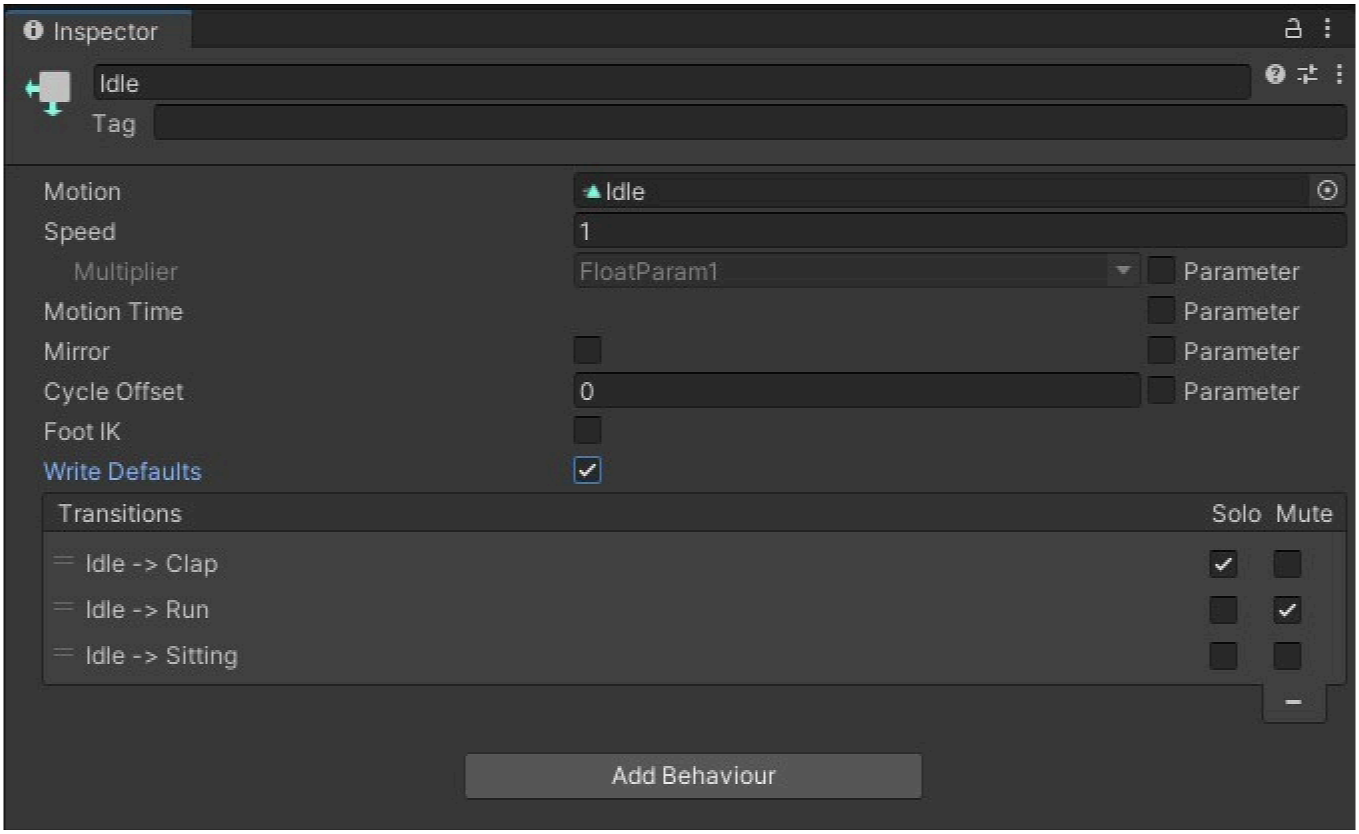
Feature Support Tables

Animator Controller Layer



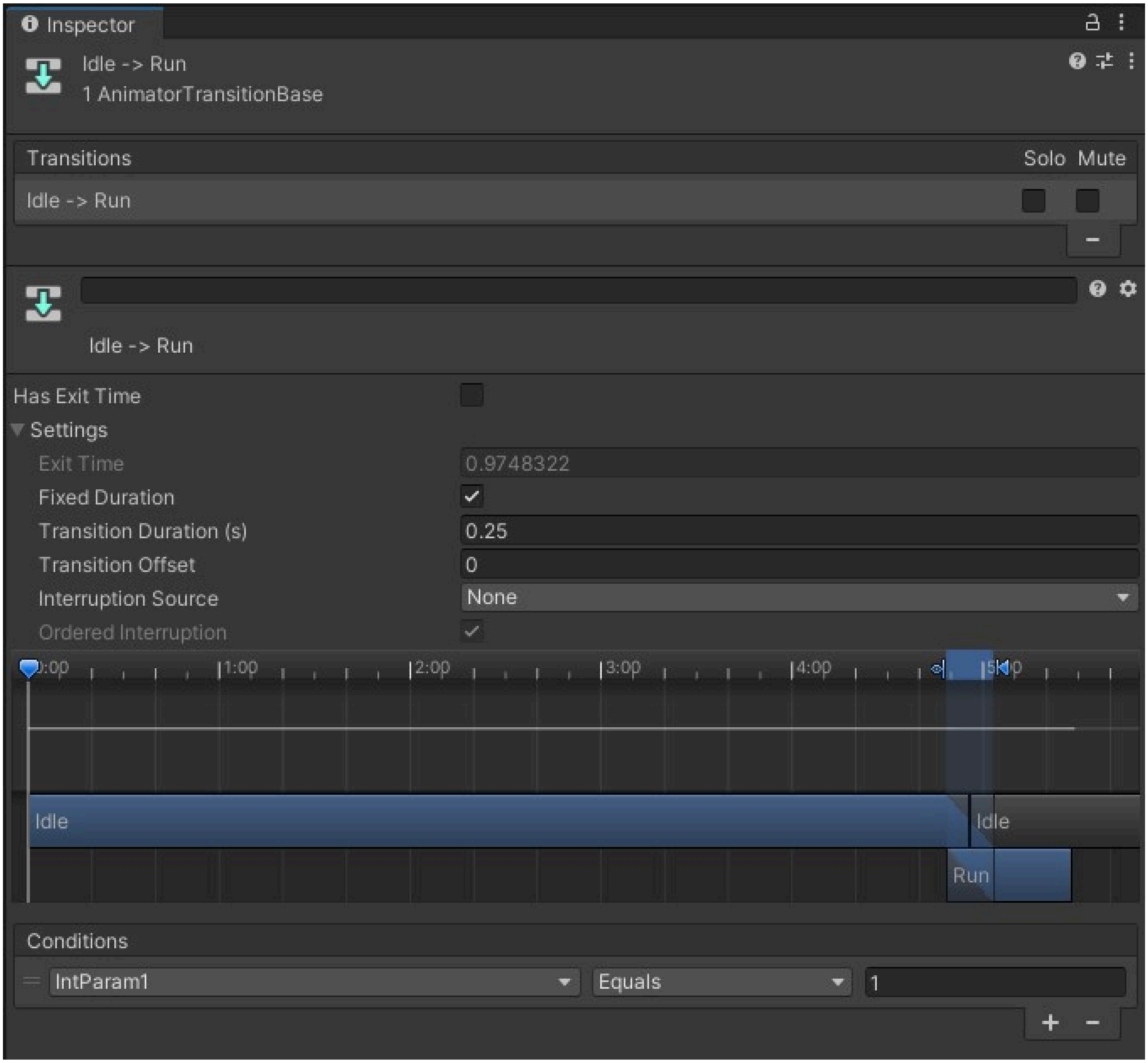
Feature Name	Support Status	Additional Notes
Multiple Layers	✅	
Sub-State Machines	✅	
Weight	✅	
Mask	✅	
Override Blending	✅	
Additive Blending	✅	
Sync	✅	With "Timing" checkbox functionality
IK Pass	❌	**Rukhanka** has own IK implementation

Animator State



Feature Name	Support Status	Additional Notes
Motion	✅	
Speed	✅	
Speed Multiplier	✅	
Motion Time	✅	
Mirror	❌	
Cycle Offset	✅	
Foot IK	❌	**Rukhanka** has own IK implementation
Write Defaults	❌	

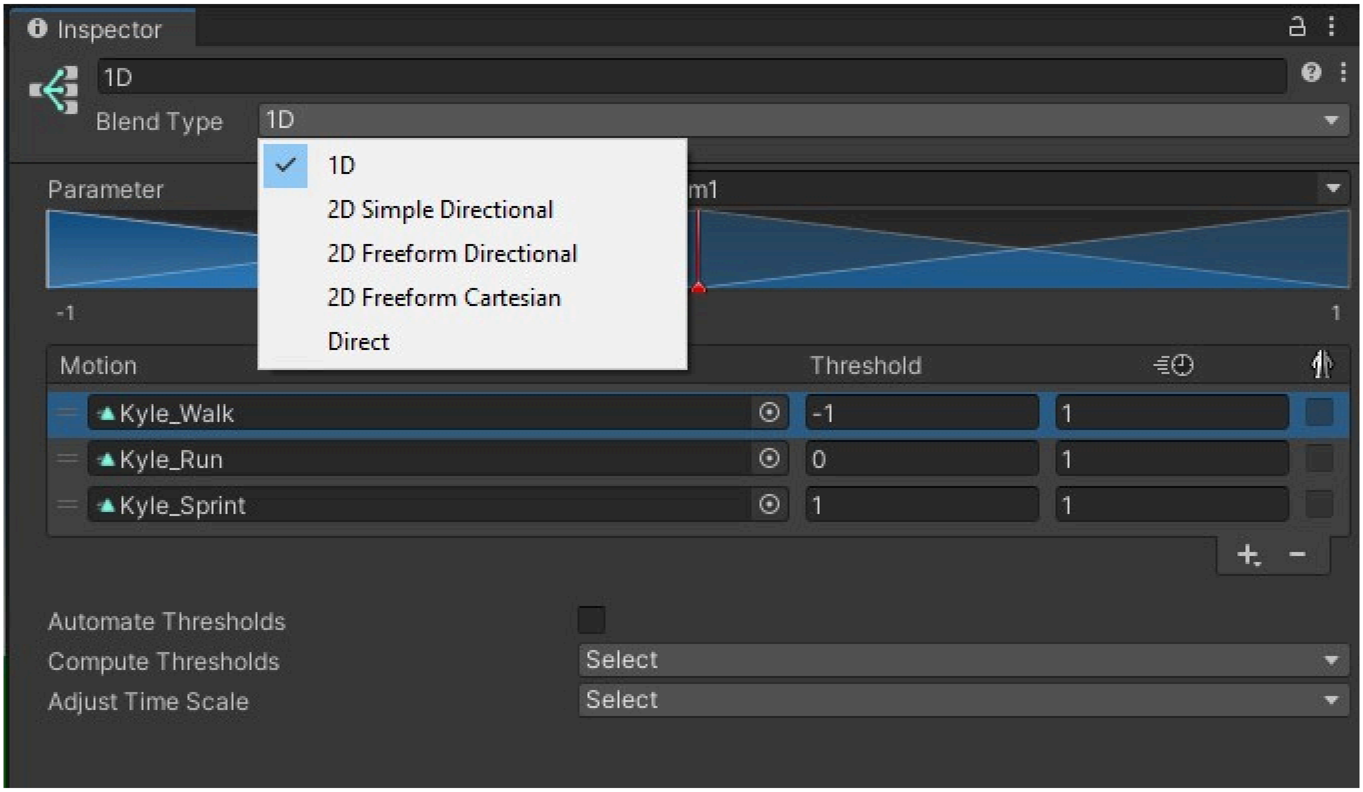
Animator Transition



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

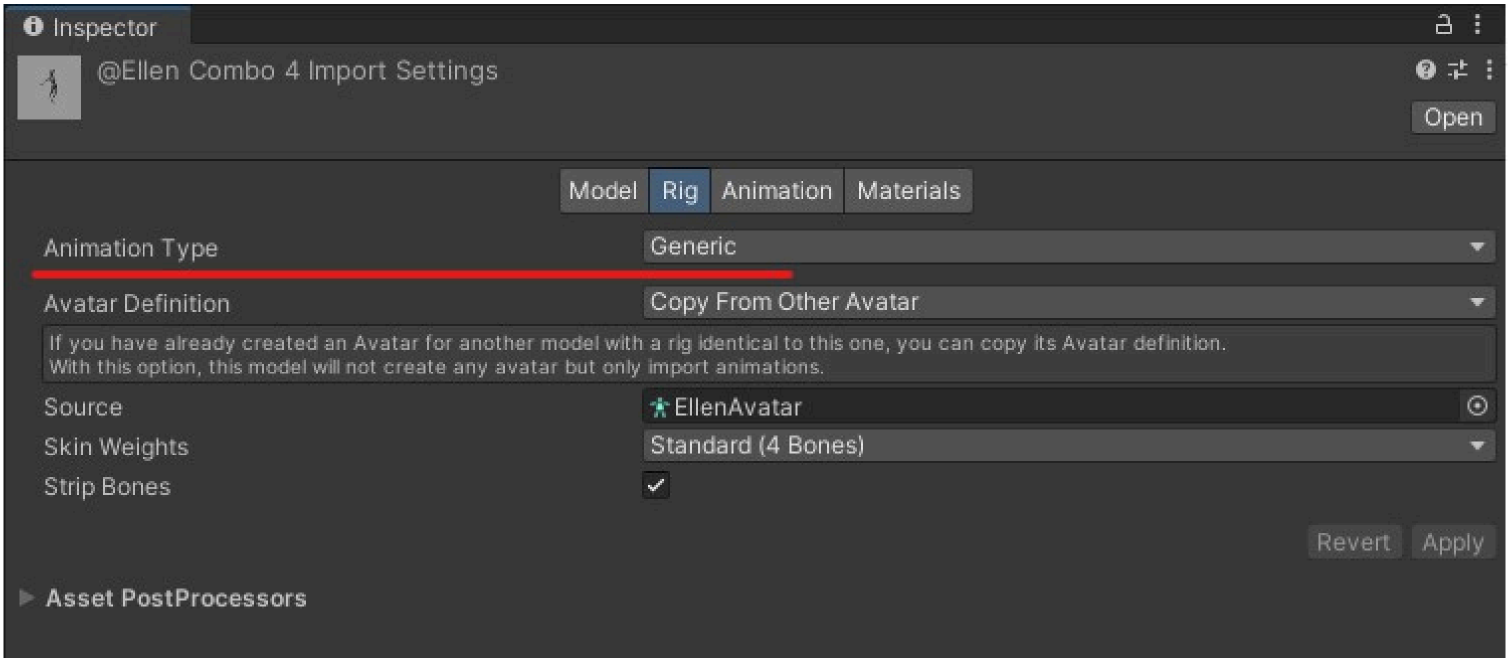
Feature Name	Support Status	Additional Notes
Transition Duration	✅	
Transition Offset	✅	
Interruption Source	✅	
Ordered Interruption	✅	
Can Transition To Self	✅	Available only in `Any State`
Int Conditions	✅	
Float Conditions	✅	
Bool Conditions	✅	
Trigger Conditions	✅	

Blend Tree Features



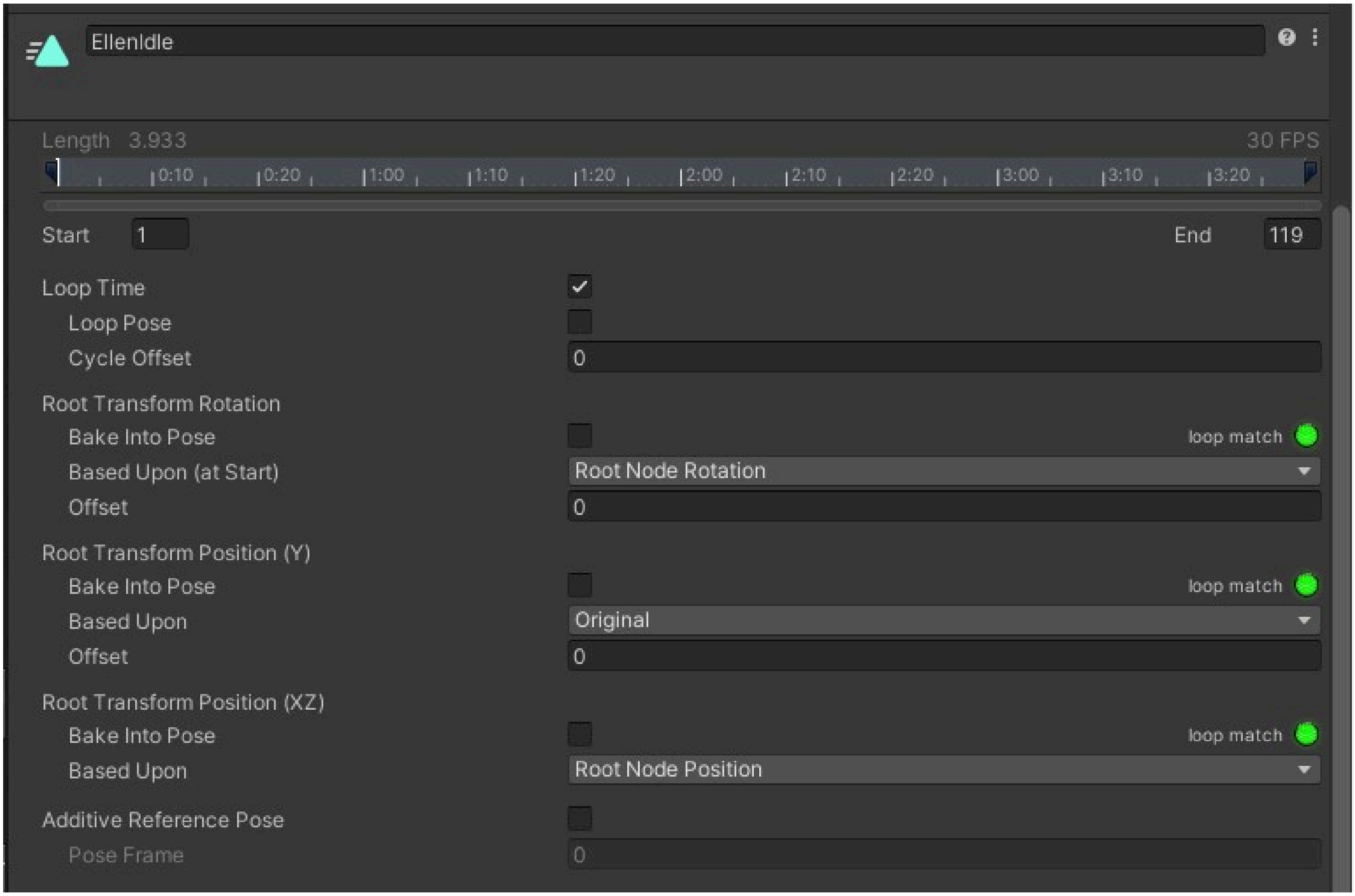
Feature Name	Support Status	Additional Notes
1D	✅	
2D Simple Directional	✅	
2D Freeform Directional	✅	
2D Freeform Cartesian	✅	
Direct	✅	
Automate Thresholds	◯	Not handled by **Rukhanka**
Compute Thresholds	◯	Not handled by **Rukhanka**
Adjust Time Scale	◯	Not handled by **Rukhanka**

Animation Rig Properties



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

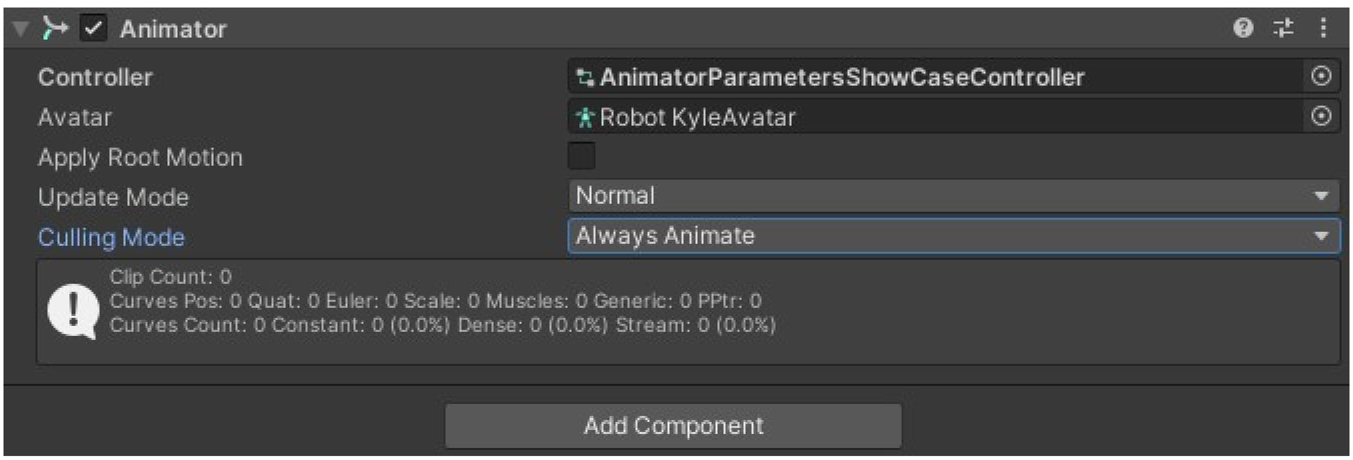
Animation Properties



Feature Name	Support Status	Additional Notes
Loop Time	✅	
Loop Pose	✅	
Cycle Offset	✅	
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	✅	Documentation
Mask	❌	
Motion	✅	Documentation

Animator Features



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

Frequently Asked Questions (FAQ)

Skinned mesh is corrupted after proper setup

Sometimes Unity's mesh metadata still contains incorrect data even after proper rig setup. This often happens with models downloaded from asset store and other third-party sources. In this case, full mesh reimport is needed. Navigate to the folder with the mesh fbx file and delete the corresponding metafile. I.e. if mesh is `Assets/my_skinnedmesh.fbx`, then `Assets/my_skinnedmesh.fbx.meta` file should be deleted and the whole rig setup process should be repeated.

Bone attachment visuals are looking corrupted

If your bone attachments are ordinary meshes, make sure that the shader used for their materials is **not** deformation compatible (it does not contain a `Compute Deformation` shader graph node).

GPU bone attachments disappear after assigning a special GPU attachment shader to it.

The GPU attachment shader must have access to the rig data. This is done by adding the `GPUAttachmentAuthoring` component to the GPU attachment GameObject in authoring mode. Without this component, GPU attachments will be placed incorrectly and can disappear completely. More info is on the Bone Attachments page.