

GPU ECS ANIMATION BAKER

[Overview](#)

[Installation](#)

[Requirements](#)

[Installation guide](#)

[Asset contents overview](#)

[Limitations](#)

[QuickStart](#)

[1. Adapt your materials/shaders](#)

[2. Attach baker component](#)

[3. Configure & Generate](#)

[Configuration Settings](#)

[1. Animations](#)

[2. Generate AnimationIds Enum](#)

[3. Use Predefined Animation EventIDs + list of those event IDs](#)

[4. Generate AnimationEventIds Enum](#)

[5. Attachment Anchors](#)

[6. Generate Attachment Anchor Ids Enums](#)

[7. Bone Usage](#)

[8. Transform Usage Flags Parent & Children](#)

[Generation process](#)

[3. Use the GpuEcsAnimator object](#)

[Advanced topics](#)

[Blending](#)

[Attachments](#)

[Animation Events](#)

[Using predefined animation events](#)

[GPU Instancing](#)

[Using baking services from your own editor code](#)

[Support](#)

Overview

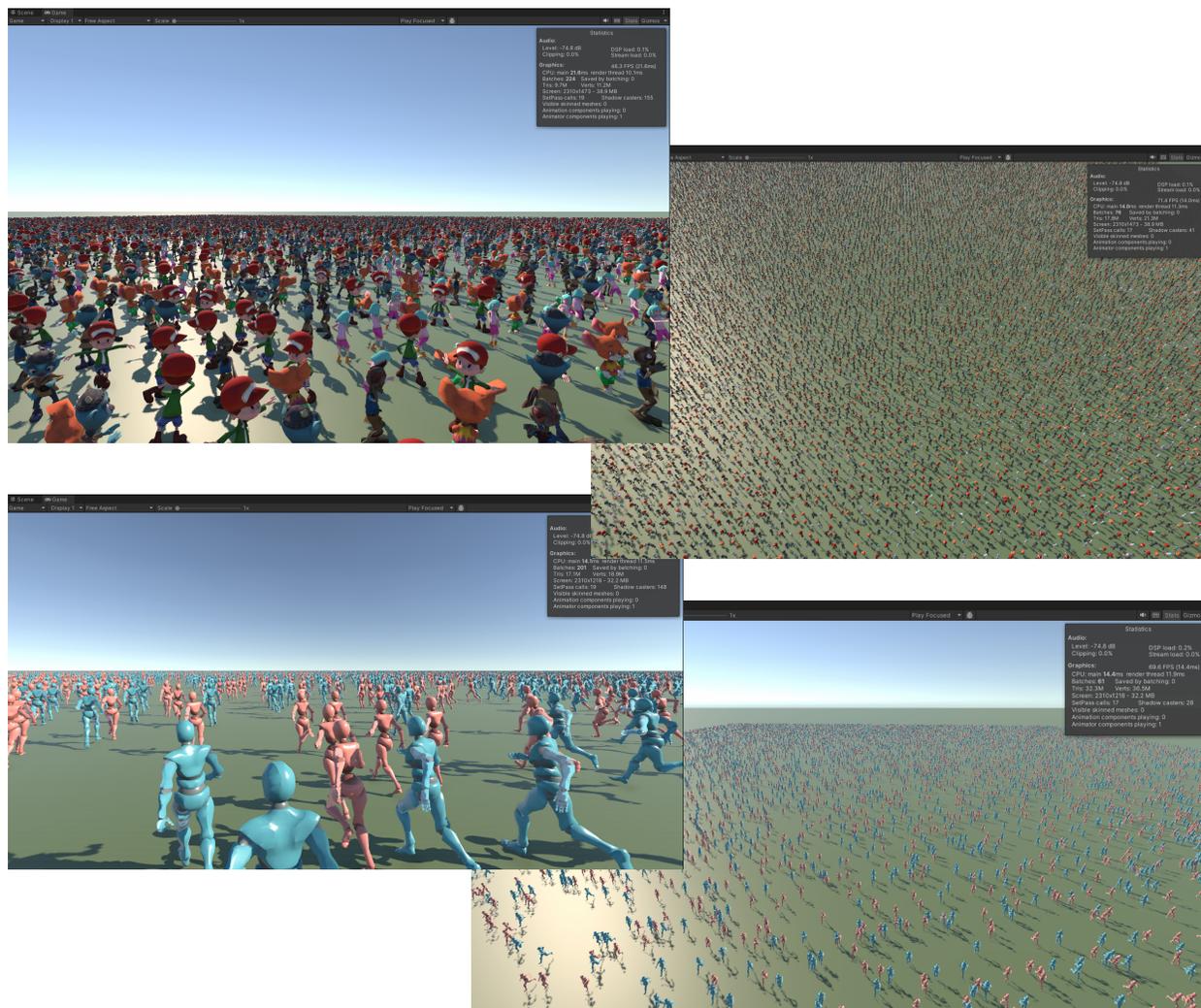
GPU ECS Animation Baker combines the power of DOTS ECS with GPU Animation instancing to allow you to animate tens of thousands of characters at the same time, each with different animations.

It works by baking all vertex bone weights into the UV channels of the meshes (uv1, uv2 & uv3) + baking all the bone transforms per frame into animation textures. A custom vertex shader will take care of the rest. An ECS animation system takes care of updating the shader values each frame.

The custom vertex shader is implemented as a ShaderGraph Subgraph, so it can be easily integrated in any of your existing ShaderGraph shaders.

The baker supports URP & HDRP, LODs, blend sampling & animation transitions.

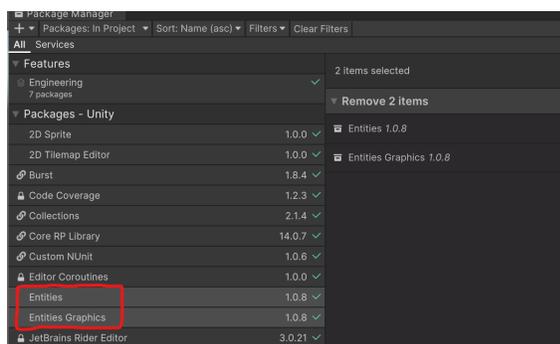
Using this technique, you can easily have tens of thousands individually animated characters on screen.



Installation

Requirements

GPU ECS Animation Baker v.1 requires Unity 2022.2.20f1 or later with ECS 1.0.8 or later. It requires both the 'Entities' as the 'Entities Graphics' package:



The asset will be kept up to date to stay compatible with the latest ECS version.

The package fully supports HDRP & URP. However, all sample scenes were made in URP, so it's best to learn about the package within a URP test project if you are planning to use it in an HDRP project. It's easy to convert the sample scenes to HDRP though. In the Samples/Shaders folder both the URP & HDRP variation of the sample shaders used in the sample scenes are provided.

Installation guide

After downloading the asset, make sure you import at least everything from the Engine folder. This is enough to make use of it, although the Samples folder is essential to learn about the tool.

Make sure you have already the packages Entities & Entities Graphics into your project via the package manager.

It's best to change the rendering path to URP Forward+. Entities Graphics requires this. You will get a warning if you don't.

If you want to run the sample scenes Menu, make sure you add all the sample scenes to the build menu:

Scenes In Build	
✓ GPU ECS Animation Baker/Samples/SampleScenesMenu	0
✓ GPU ECS Animation Baker/Samples/SampleScenes/0_Basics/0_Basics	1
✓ GPU ECS Animation Baker/Samples/SampleScenes/1_LODs/1_LODs	2
✓ GPU ECS Animation Baker/Samples/SampleScenes/2_Transitions/2_Transitions	3
✓ GPU ECS Animation Baker/Samples/SampleScenes/3_AnimatorParameters/3_AnimatorParameters	4
✓ GPU ECS Animation Baker/Samples/SampleScenes/4_BlendSampling/4_BlendSampling	5
✓ GPU ECS Animation Baker/Samples/SampleScenes/5_Attachments/5_Attachments	6
✓ GPU ECS Animation Baker/Samples/SampleScenes/6_Events/6_Events	7
✓ GPU ECS Animation Baker/Samples/SampleScenes/Festival/Festival	8
✓ GPU ECS Animation Baker/Samples/SampleScenes/Marathon/Marathon	9

You're now ready to use GPU ECS Animation Baker!

Asset contents overview

The assets consist of the following directories:

- **GPU ECS Animation Baker**
 - **Engine**

This contains all the code to make the baker work (editor + runtime)
 - **Samples**
 - **Characters**

All the character models, materials, textures & baked animations objects
 - **SampleScenes**

Sample scenes are ordered in complexity.
Festival & Marathon are big crowd demo scenes
SampleScenesMenu allows you to browse all the sample scenes.

The best way to learn is to follow the tutorials, examine the example scenes & read this documentation for details.

Tutorials: <https://www.youtube.com/playlist?list=PLerop1JzrAobUF2d8lafj5waqSq3lb8k1>

Demos: <https://www.youtube.com/playlist?list=PLerop1JzrAoa8Vpz1p3Ps8EGRduhaDIP3>

Limitations

Disclaimer: The GPU ECS Animation baker is built for performance, and excels in animating many characters at once, but it is in no way as versatile & feature-rich as the existing Unity Animator/Skinned Mesh Renderer solution. There are a few things to bear in mind:

- You can only use UV0 in your meshes. UV channels 1,2 & 3 are used by the baker, so GPU ECS Animation baker doesn't support models that use these channels.
- Your materials need to use ShaderGraph shaders. It would be possible to implement the baker for coded shaders, but this falls outside the current scope of implemented

features. Check out the shader code at `/GPU ECS Animation Baker/Engine/Shader` if you want to implement this yourself.

- All classic Animator features are supported insofar as they can be baked and sampled. The classic Animator itself however does NOT run at runtime at all, so be aware of this. This also means that any feature that necessarily requires runtime calculation of the bone structure (like IK) is & can never be supported by the baker.
- The package fully supports HDRP & URP. However, all sample scenes were made in URP, so it's best to learn about the package within a URP test project if you are planning to use it in an HDRP project. It's easy to convert the sample scenes to HDRP though. In the Samples/Shaders folder both the URP & HDRP variation of the sample shaders used in the sample scenes are provided

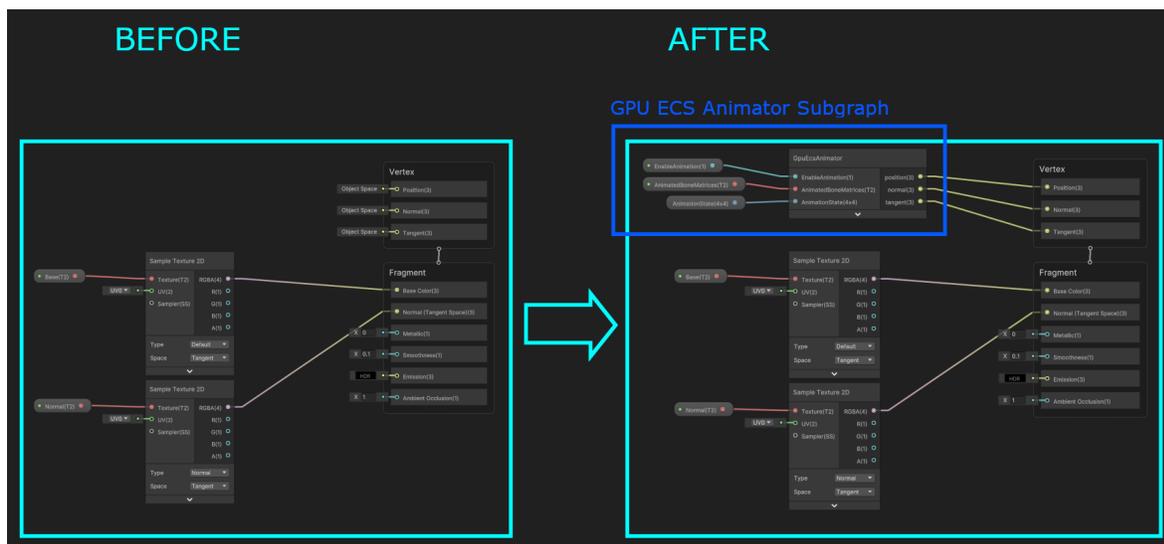
QuickStart

Everything starts with a classic game object with an animator and 1 or more SkinnedMeshRenderer(s) and optionally an LODGroup. The baking process will start from this object.

1. Adapt your materials/shaders

The first thing to do is to extend the shaders that are used in your SkinnedMeshRenderer with our special ShaderGraph GpuEcsAnimator Subgraph.

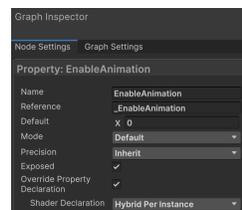
Let's say you have a simple shader URP Lit shader with a base & normal map as input. If you want to make this shader compatible with GPU ECS Animation, you simply have to add the GpuEcsAnimator like this:



The shader requires 3 extra parameters to be defined on your shader

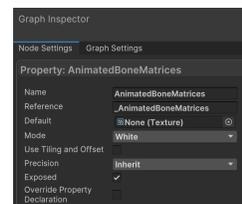
1) EnableAnimation

This is a boolean that allows normal viewing of the original asset by disabling the animation shader on it. Animation will be enabled automatically in the ECS baker/convertor. It needs to be exposed & 'Override Property Declaration' must be checked & set to 'Hybrid Per Instance'



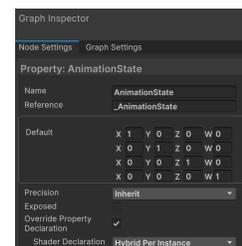
2) AnimatedBoneMatrices

This texture contains all the animation data. Each row represents a frame of an animation & each 4 columns represent a mesh bone. It needs to be exposed.



3) AnimationState

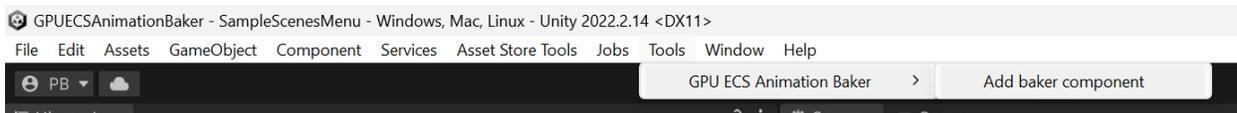
This will drive the animation state at runtime. 'Override Property Declaration' must be checked & set to 'Hybrid Per Instance'



Note: The simplest way to do this is to simply copy paste the 3 parameters & the subgraph from one of the sample shaders found in /GPU ECS Animation Baker/Samples/Shader.

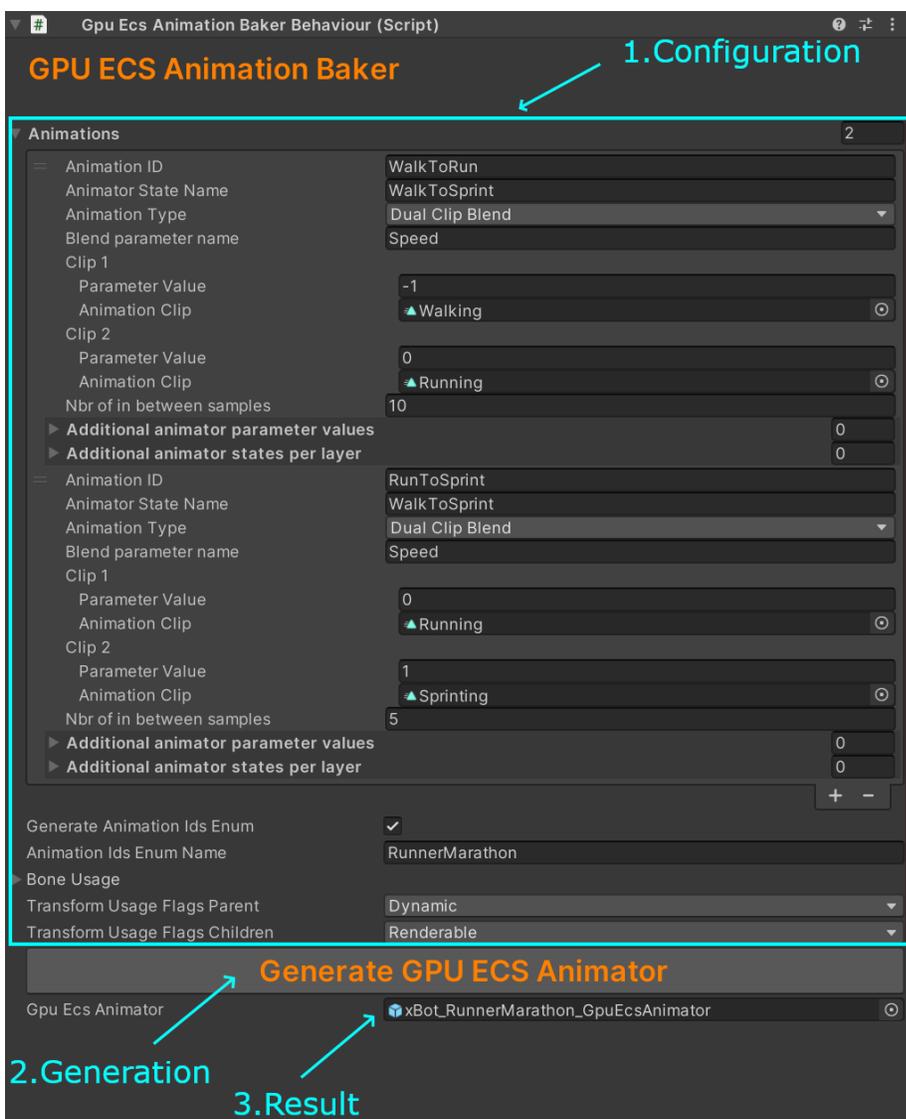
2. Attach baker component

Attach the baker component through the menu when the Animator game object you wish to bake a GPU ECS Animator for, is selected.



3. Configure & Generate

You can now configure all baking settings, and when done generate the GPU ECS Animator object.



Configuration Settings

1. Animations

Configuration of all animations you want to bake

- **Animation ID**
Only used when generating enum code file to identify animations (see below)
- **Animator State Name**
The Animator state name that will be used during sampling (regardless what layer it is on)
- **Animation Type**
This can be either of these 2:
 - 1) Single Clip
A single clip animation just plays a single (non-blended) animation
 - **Animation Clip**
Only used to determine the length of the animation(*)
 - 2) Dual Clip Blend
Dual clip blends will allow you to blend between 2 animations controlled by an animator parameter. The baker will sample this animation across a range of values for this parameter
 - **Blend parameter name**
The animator parameter that will be used to blend
 - **Clip1, Clip2**
 - **Parameter Value**
The animator parameter value to blend from or to
 - **Animation Clip**
Only used to determine the length of the animation(*)
 - **Nbr of in between samples**
The nbr. of samples to take. More will result in better results, but larger baked animation textures
- **Loop**
Boolean to indicate that the animation needs to loop or not
- **Additional parameter values**
Any additional Animator parameters that will be used during sampling (independent of blending)
Specify name, type & value for each.
- **Additional animator states per layer**
Any additional Animator states that need to be set on different layers
Specify layer index & animator state name for each.

(*) It's important to realise that what matters for the generation of the animation textures is not the actual animation clips that are set, but rather the animator state (+ additional animator states per layer). The animation clip just needs to be provided as a reference to determine the length of the sample to be baked.

2. Generate AnimationIds Enum

It can be handy to use an enum file to control animation from code in a more user-friendly way.

If you don't generate an enum, you will have to control which animation to play by its index

It will also create a handy initializer behaviour which you can attach to the baked Gpu Ecs Animator, to initialise an animator without any additional code.

3. Use Predefined Animation EventIDs + list of those event IDs

If you don't check this box, the baker will find all events inside your animations and build a list of IDs itself based on what it finds. You can however provide a predefined list (see Advanced Topics -> Animation Events)

4. Generate AnimationEventIds Enum

It can be handy to use an enum file to identify events from code in a more user-friendly way, similar to the Animation Ids enum. If you use predefined animation event IDs, you have direct control over what will be generated.

5. Attachment Anchors

Configuration of all attachment anchors you want to bake

- **Attachment Anchor ID**
Only used when generating enum code file to identify attachment anchors (see below)
- **Attachment Anchor Transform**
This needs to point to a gameobject inside the bone hierarchy of your animator.

6. Generate Attachment Anchor Ids Enums

It can be handy to use an enum file to identify anchors from code in a more user-friendly way, similar to the Animation Ids enum. It will also create a handy initializer behaviour which you can attach to attachments so they'll stay attached to the parent entity in the correct place without any additional code.

7. Bone Usage

You should specify a default number of bones per vertex to take into account. More bones will yield better results (more accurate vertex positioning) but requires more from the GPU at runtime.

You can override the amount of bones per LOD to further optimise performance.

8. Transform Usage Flags Parent & Children

These flags will be used in the ECS entity baker for both the Animator (parent) as the children (Meshes). Check documentation for TransformUsageFlags enum in the Unity ECS Baking documentation.

Generation process

The generator will create a new directory (if it doesn't already exist) with the prefix 'BakedAssets_'+ the animator gameobject name. Inside it will generate a bunch of files:

- The enumeration & initializer C# files to identify animations, events & attachments (optional)
- Animation textures
- Copies of the original materials with the animation textures assigned to it
- Mesh copies with the vertex bone weights baked into UV coordinates uv1, uv2, uv3
- The GpuEcsAnimator object, which can be baked by the Unity ECS baker into an entity

3. Use the GpuEcsAnimator object

If you examine the GpuEcsAnimator you will see that it will mirror the structure of the original Animator gameobject. All SkinnedMeshRenderer components will have been replaced with regular MeshRenderer components with the generated materials & meshes assigned to them.

Any LODs will have been converted. Monobeaviours with ECS conversion bakers will be attached.

The simplest way to use the entity is by putting it in a Subscene (which will tell Unity it needs to be converted to an ECS entity). In most real game use cases though, you will use them as prefabs for ECS entity spawner systems.

The animation will be controlled by 2 systems:

- GpuEcsAnimatorSystem
This system will prepare the data to be sent to the shaders
- GpuEcsAnimatedMeshSystem
This system will populate all material overrides with the prepared data

The only thing you need to do as a user to control these animations is to set the GpuEcsAnimatorControlComponent component on the entity you want to animate:

```
using Unity.Entities;

namespace GPU ECS Animation Baker.Engine.AnimatorSystem
{
    public struct GpuEcsAnimatorControlComponent : IComponentData
    {
        public AnimatorInfo animatorInfo; // All info about the animation you want to play
        public float startNormalizedTime; // An option to start the animation from an arbitrary
        position (0 to 1)
        public float transitionSpeed; // The transition speed that will be applied when switching
        to another animation
    }

    public struct AnimatorInfo
    {
        public int animationID; // the unique animation ID, can be assigned from the generated
        enum file
        public float blendFactor; // From 0 to 1, going from clip1 to clip2
        public float speedFactor; // <1 to make the animation go slower, >1 to make it go faster
    }

    public enum GpuEcsAnimatorControlStates
    {
        Start,
        Stop,
        KeepCurrentState
    }

    public struct GpuEcsAnimatorControlStateComponent : IComponentData
    {
        public GpuEcsAnimatorControlStates state;
    }
}
```

Typically you will set this component whenever you want to transition to another animation or when you want to change the parameter of a running animation (blendFactor or speedFactor).

The GpuEcsAnimatorControlStateComponent can be set to start a stopped animation or to stop a running animation. An animation will stop automatically when it's not set to loop and has ended its cycle time.

Alternatively, for convenience there is also an GpuEcsAnimatorAspect aspect class that you can use to control the animation:

```
using Unity.Entities;
namespace GPU ECS Animation Baker.Engine.AnimatorSystem
{
    public readonly partial struct GpuEcsAnimatorAspect : IAspect
    {
        private readonly RefRW<GpuEcsAnimatorControlComponent> control;
        private readonly RefRW<GpuEcsAnimatorControlStateComponent> controlState;

        public void RunAnimation(int animationID,
            float blendFactor = 0f, float speedFactor = 1f, float startNormalizedTime = 0f, float
            transitionSpeed = 0f)
        {
            control.ValueRW = new GpuEcsAnimatorControlComponent()
            {
                animatorInfo = new AnimatorInfo()
                {
                    animationID = animationID,
                    blendFactor = blendFactor,
                    speedFactor = speedFactor
                },
                startNormalizedTime = startNormalizedTime,
                transitionSpeed = transitionSpeed
            };
            StartAnimation();
        }

        public void StartAnimation()
        {
            controlState.ValueRW = new GpuEcsAnimatorControlStateComponent() {
                state = GpuEcsAnimatorControlStates.Start
            };
        }

        public void StopAnimation()
        {
            controlState.ValueRW = new GpuEcsAnimatorControlStateComponent() {
                state = GpuEcsAnimatorControlStates.Stop
            };
        }
    }
}
```

There is also another convenient initializer monobehaviour script that allows you to initialize the correct animation at baking time:

```
namespace GPUAnimationBaker.Engine.AnimatorSystem
{
    [RequireComponent(typeof(GpuEcsAnimatorBehaviour))]
    public class GpuEcsAnimatorInitializerBehaviour<T> : GpuEcsAnimatorInitializerBehaviour where
    T : Enum
}
```

If you define a new behaviour that subclasses this `GpuEcsAnimatorInitializerBehaviour` with the correct animation enum type, you can attach it to the animator object and select the animation to initialize at design time, so it will be initialized already during baking. This initializer script will be generated along with the enum file.

Similarly an initializer script also exists for attachments (see attachments)

Best way to learn how to control the animations is to look at the samples provided with the asset where each of these methods described above are used.

Advanced topics

Blending

The classic Unity animator does blending via interpolation of the bone structure relative angles. This is however not possible when using GpuEcsAnimationBaker, because each vertex position is calculated in the shader based on bone matrices that are pre-baked per frame. Interpolation instead happens between vertex positions. It's important to note that this can cause unwanted mesh deformations when the difference between the 2 blended poses is large. Let's examine the 3 scenarios where vertex interpolation blending will be applied:

1) Transitions

Transitions between 2 different animations typically don't last more than half a second or so. In this case, the deformation is not easily spotted, especially in a context where many characters are on-screen.

2) Dual Clip Blends

When blending between 2 poses, for example between a walk & a run cycle, the blend is continuous so a simple vertex interpolation between the walk & run pose will not look ok. However, we can fix this by sampling enough in-between poses during the baking process. Vertex interpolation will then occur between 2 very similar poses and will look just fine.

3) Frame-to-frame interpolation

Frames are sampled & baked at 30 FPS. This means that in a game, running at 60+ FPS, interpolation will occur between pre-baked vertex positions in time. These 2 poses will by definition be very similar, so again vertex interpolation will work just fine.

Attachments

Sometimes you want to attach dynamically different objects to a character (swords, pieces of clothing, tools, etc..) and you want the animation to keep working while swapping out those objects.

There are 2 ways to do this:

- You can simply add these objects and rig them together with the actual character mesh. The attached object will then be a Skinned MeshRenderer and will be treated just as the character mesh itself. The object will move through vertex displacement in the shader just as the character vertices do. This approach is simple & fast but less flexible as you cannot easily swap out the object or detach it (for example if an arrow flies away after the shooting animation is finished)

- You can define attachment anchors in the baker in the correct position inside your source bone structure. The baker will then record those anchor positions & rotations during baking. At runtime a special attachment system will look for entities that have a `GpuEcsAttachmentComponent` attached to it and will place them in the correct position based on the anchor Id defined in the component.
A convenient initializer script is available to attach this component with the right values for you.

Animation Events

With a classic Animator you have the option to define events inside the Unity animation editor & call a custom function. Obviously, these functions will not be called in the context of a baked ECS entity.

However, you can still make use of this interface by attaching a `GpuEcsAnimationEventBakerBehaviour` to your source object and then defining the events in such a way that they call the `RaiseEvent(string)` function of this behaviour.

When you do this, the baker will pick up the event data and raise events at runtime by adding an event to the `GpuEcsAnimatorEventBufferElement` dynamic buffer of the entity. You can write your own event handler to deal with these events within the context of ECS.

Each frame this buffer will be cleared before new events are added, so you need to make sure to read all events each frame not to miss them.

Using predefined animation events

In the configuration you can choose to use a predefined list of event IDs. This means that the baker, rather than making its own list based on what it finds in the animations, it will use the list you provide. This gives you direct control over the IDs and their order. If the baker finds events that are not in your list, it will ignore them with a warning during baking.

This can be handy when you want to share the same events across different characters who don't necessarily have all the events, or in the same order. In that case you can use 1 predefined list which you set for all these characters and they will raise the same events.

GPU Instancing

As all animation shaders will be based on ShaderGraph shaders, GPU instancing is supported by default. All you have to do is enable it in the material.

Using baking services from your own editor code

Instead of attaching the baker component to your animators and configuring baker settings manually in the editor + starting the process via the 'Generate GPU ECS Animator' button, you can also automate this process via your own editor code by calling the following functions:

```
namespace GPUECSAnimationBaker.Engine.Baker
{
    public static class GpuEcsAnimationBakerServices
    {
        public static GameObject GenerateAnimationObject(
            string assetPath,
            GpuEcsAnimationBakerData bakerData,
            string animatorName,
            string generatedAssetsFolder,
            string nameSuffixAsset = "_GpuEcsAnimator",
            string nameSuffixAnimationIDsEnum = "_AnimationIDs",
            string nameSuffixAnimationInitializerBehaviour = "_AnimationInitializerBehaviour",
            string nameSuffixAnimationEventIDsEnum = "_AnimationEventIDs",
            string nameSuffixAnimationAnchorIDsEnum = "_AttachmentAnchorIDs",
            string nameSuffixAttachmentInitializerBehaviour = "_AttachmentInitializerBehaviour",
            string meshPartSuffix = "Mesh",
            string animationMatricesTexturePartSuffix = "AnimationMatricesTexture",
            string materialPartSuffix = "Material")
        {
        }

        public static GameObject GenerateAnimationObjectFromModel(
            GameObject refModel,
            GameObject sourceModel,
            GpuEcsAnimationBakerData bakerData,
            string animatorName,
            string generatedAssetsFolder
            string nameSuffixAsset = "_GpuEcsAnimator",
            string nameSuffixAnimationIDsEnum = "_AnimationIDs",
            string nameSuffixAnimationInitializerBehaviour = "_AnimationInitializerBehaviour",
            string nameSuffixAnimationEventIDsEnum = "_AnimationEventIDs",
            string nameSuffixAnimationAnchorIDsEnum = "_AttachmentAnchorIDs",
            string nameSuffixAttachmentInitializerBehaviour = "_AttachmentInitializerBehaviour",
            string meshPartSuffix = "Mesh",
            string animationMatricesTexturePartSuffix = "AnimationMatricesTexture",
            string materialPartSuffix = "Material")
        {
        }
    }
}
```

GenerateAnimationObject

- **assetPath** : the asset path of the source prefab asset you want to generate a GPU ECS Animator from

GenerateAnimationObjectFromModel

- **refModel** : the instantiated reference model
- **sourceModel** : the source model
- **bakerData** : all the baker Configuration data. This structure is equivalent to what you see in the editor when you attach a baker component to a prefab
- **animatorName**: the name that will be used as a prefix for naming all the generated files (*"{animatorName}_GpuEcsAnimator, {animatorName}_AnimationMatricesTexture_{name of skinned meshrenderer}, etc..)*
Note: When you use the editor UI to generate the GPU ECS Animator, this will be just the name of the original asset
- **generatedAssetsFolder** : the folder that will contain all the generated assets during baking.
Note: When you use the editor UI to generate the GPU ECS Animator, this will be a subfolder in

the folder of the source prefab called 'BakedAsset_{*name of asset*}'. Using this function will allow you to choose your own folder

- **Naming suffixes** : these are all strings that allow you to fully control the naming of all the various generated files if you wish to do so. The defaults should be fine for most cases though.

This way you can write your own generator for multiple animators in your project at once, sharing all or part of the configuration and deciding yourself where all the assets end up.

Support

<http://www.headfirststudios.com/theorange coder>

<https://discord.gg/DnBzvMC2uG>

Contact me at:

pbosteels@gmail.com