User Manual

1.0 Introduction

This manual is intended to provide details on every feature of 'Motion Matching for Unity' (MxM). If you haven't already, check out the Quick Start Guide first. There are also a number of video tutorials which supplement this manual.

1.1 Support

The best place for direct support is the 'Motion Matching for Unity' discord channel. Once you join, send me a private message with your asset store invoice number and I will give you 'verified status' where you can obtain the highest level of support.

- Discord
- Unity Forum
- Unity Connect

Note: While I do provide some level of support on the Unity Forum, Unity Connect and through email, it may take me longer to respond through these channels. I highly recommend using the Discord instead.

1.2 Bug Reports

Bug reports should be submitted to the github issue tracker for MxM. Please read and follow the instructions on the github page when submitting a bug report. Bug reports with lacking detail will be ignored.

If you are unsure if something is a bug or not, feel free to ask on the Discord channel.

1.3 Tutorials and Videos

There are several tutorials and videos available for MxM that supplement this manual. They are linked below:

- Quick Start Guide
- Video Tutorials

Note: MxM is continually changing and evolving. Unfortunately it is not possible to update the video tutorials for each update so there may be differences between what is shown in the tutorials and the exact state of MxM.

1.4 Standalone Demo

There are two standalone demo's available to download and play for free. These demo's show the animation output you can expect from MxM. The standalone demo's available include:

- Standalone Stress Test (MxM v2.1)
- Standalone Demo (MxM v1.7b)

The level and assets in the standalone demo are NOT included with MxM. However, the standalone stress test scene is included.

Note: The ‘Standalone Demo’ was built with MxM v1.7 beta and is very out of date. This demo will be updated soon.

1.5 Who Is This Asset For

This asset is for anyone who wants to have fluid character animation in their games without the complex logic and rigidity of state machines. However, results may vary depending on your available resources animation data and coding ability.

Studios with mocap capabilities and high animation bandwidth will reap the greatest benefits from this asset. Motion matching works best with lots of mocap data captured in a specific way. This does not mean it does not work with cut clips but the results may not be as good and there are some specific requirements.

Motion matching is also most powerful if you are able to code. As a minimum you need to be able to call functions and set parameters from the MxMAnimator component. However, if you want to make the most out of MxM you will need to be able to code a custom gameplay model to control your character which is able to predict its future based on player input with some degree of accuracy. MxM will try to match your gameplay future predictions but if the gameplay predictions are poor there is only so much MxM can do.

1.6 What is ‘Motion Matching for Unity’

‘Motion Matching for Unity’ (MxM) is a motion matching animation system that replaces traditional animation state machines such as mecanim in Unity. In essence it takes a series of inputs and outputs, and then chooses an animation pose each frame for the character. At this high level, it is no different to mecanim. However, the magic comes with how it achieves this and the results.

Unlike mecanim, motion matching does not require an animation state machine with complex logic. You don’t have to define transitions and conditions or even cut clips from mocap. In the simplest form, it takes the input from the player and the current animation pose, and it searches through all animation data to find the best point to jump to. Motion Matching jumps to any point in your animation database at any time in order to best match your desired motion while still providing fluid animation.

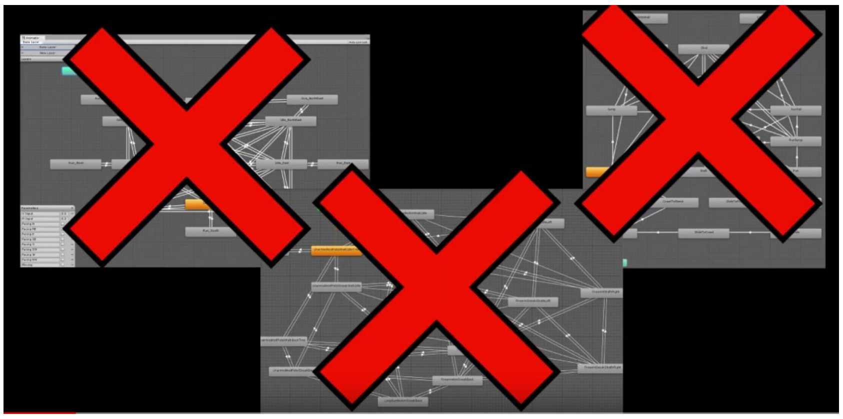


Figure 1.1 'No complex state machines'

The end result is a character with automatic turns, starts, stops, plants, accelerations, decelerations etc. without having to code complex logic like foot tracking. You never have to decide when a transition starts or ends, the system picks the best match for you. It doesn't matter if you are half way through a step, it will still pick the best animation to do a plant or stop, balancing both fluidity and responsiveness.

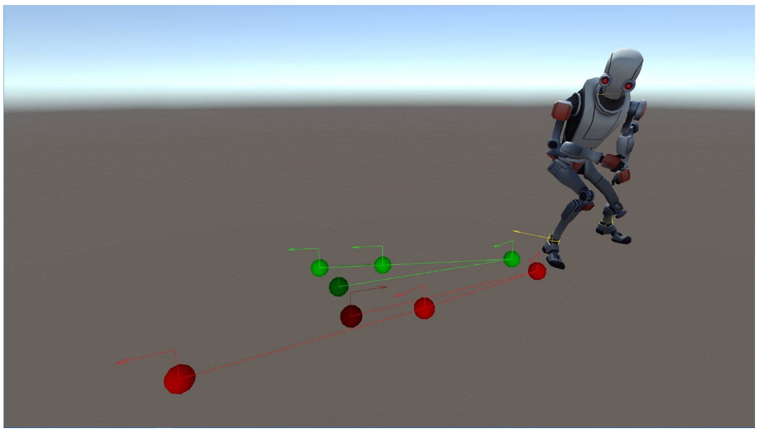


Figure 1.2 'Automatic starts, stops, turns and plants'

Many people believe that motion matching can only be used for simple locomotion. This is completely false. If your gameplay code can predict a desired future then the motion matching system can match it. This along with a dedicated idle system, events system and tagging system makes motion matching very flexible to control animation for any kind of gameplay.

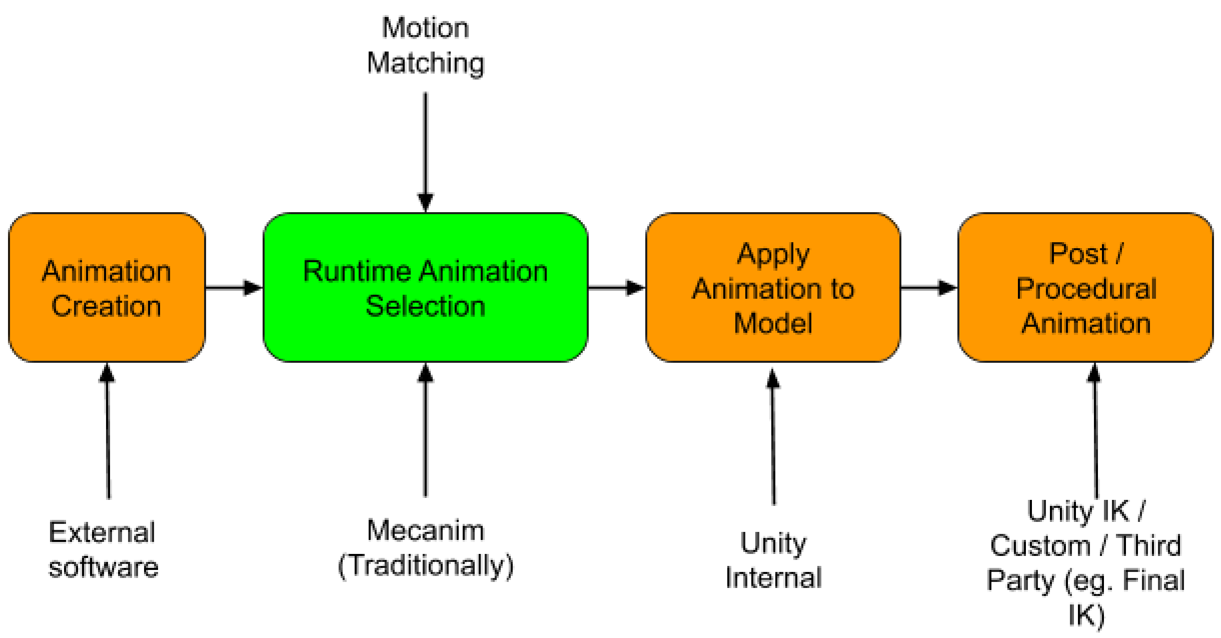


Figure 1.3 'MxMs Place in the animation pipeline'

1.7 What Motion Matching is not

Motion Matching does not 'synthesis' animation from nothing. The animations used by Motion Matching all come from a user defined set and it will never play any animation outside of this. The only variance that may occur from your animation data is the smoothing between jumps or user imposed procedural animations / IK.

MxM is not a gameplay controller. MxM's job is to output an animation based on your gameplay and animation set. If you cannot code the gameplay to tell MxM what your future goal is then it won't work. MxM comes with a very locomotion trajectory model, but you may want to create your own gameplay model. It is impossible for me to predict your gameplay, making it pointless to include too much gameplay code in MxM.

MxM does not handle IK or procedural animation. MxM generates an animation. Procedural animation and IK takes place after this step. As a result, IK solutions such as Unity's default IK system, FinalIK or 'Unity Animation Rigging' work over the top of MxM just like they work with mecanim state machines.

Note: Unity's Animation Rigging package requires an integration script which can be downloaded from the support Discord channel

MxM is not a 'Make Animation Button', 'Make Character Button' or a 'Make Game Button'!

1.8 Real World Examples

While motion matching is a relatively new technology, it has been around for a few years and some games have even shipped with it. Most notably ‘For Honor’ by Ubisoft, which was the first game to ever ship with motion matching. Since then a number of Ubisoft and EA Sports titles have shipped with motion matching systems. The soon to be released ‘The Last of Us Part 2’ also uses a motion matching system.

‘Motion Matching for Unity’ is not affiliated with any of these projects or companies but is based on the techniques made public by Ubisoft in 2016. Motion matching is a recent evolution of game animation and it is becoming very popular in the industry.

1.9 The LAW of Motion Matching

I previously stated that motion matching can be used to produce any kind of animation. This statement holds true provided you feed it with the correct inputs. The primary inputs being the animations themselves and the gameplay prediction model.

Animations need to provide a decent range of coverage (to be discussed in detail later) and you may get poor results if you are missing transitions.

MxM will try to best match your gameplay prediction model. However, if your prediction model is poor then the resulting animation will also be poor.

Therefore the law of Motion Matching is this:

Trash In = Trash Out

2.0 Workflow

The general workflow of motion matching is simple:

1. Create a PreProcessor, configure it and add animations
2. Generate an MxMAnimData asset from the pre-processor
3. Add an MxMAnimator component to your character
4. Add the MxMAnimData asset to you MxMAnimator
5. Hook up your custom gameplay prediction model to the MxMAnimator through the ITrajectoryGenerator interface.
6. Play!

Of course this is a very broad overview and there is plenty of finessing that can be done at each of these stages to modify animation behaviour and improve outputs. I will go into detail on all aspects of this throughout this manual.

2.1 The Pre-Processor (MxMPreProcessData)

Every motion matching character starts with the pre-processor. The pre-processor is where you configure your data and provide most of the static inputs (non runtime input) for the motion matching system including:

1. Trajectory Points (number and timing)
2. Which character joints to match
3. Tag and event names / definitions
4. Animations

2.1.1 Create a Pre-Processor

To create a pre-processor, right click in your project view and choose MxM / Core / PreProcessData as shown in figure 2.1 . This will create a custom asset which, when selected, will display its own inspector.

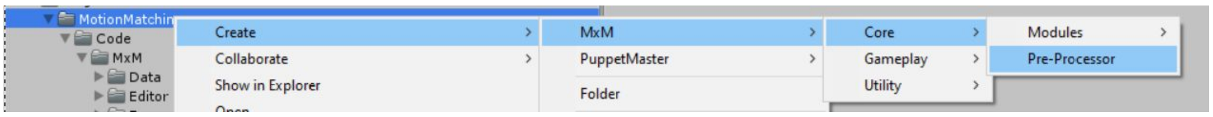


Figure 2.1 'Creating a Pre-Processor'

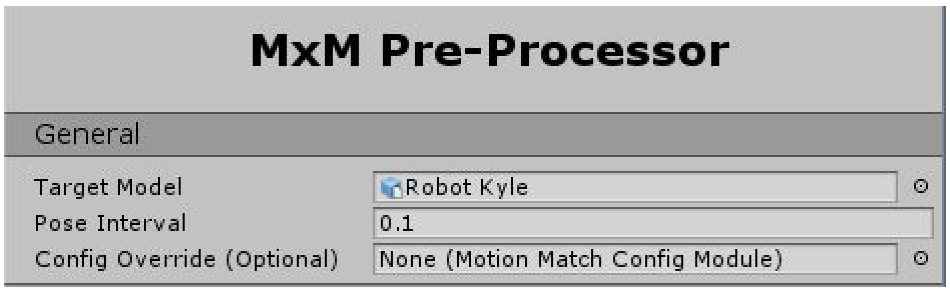
2.1.2 Basic Settings

There are two basic settings that need to be set as well as an optional property. These include the 'character model', to use for the pre-processing stage, and the 'pose interval'.

Target Model - the target model (not prefab) is the character model that will be used when the pre-processor is running it's calculations. It's preferable to use the model for the character you want to use motion matching with. This is because the characters with different proportions and sizes will have differing root motion. However, provided characters are somewhat similar, there shouldn't be an issue with sharing anim data

Pose Interval - The pose interval is the time between recorded poses. The pre-processor will generate a pose (PoseData) for every pose interval in your animation data. The default of 0.1s is recommended for most cases where there is sufficient data. In some cases, particularly when lacking animations or using cut clips, the 'Pose Interval' could go as low as 0.05s.

Higher pose interval values will improve performance but reduce quality. Lower values will reduce performance and might have questionable improvements in quality. Setting the value too low may also result in less responsive gameplay.


MxM Pre-Processor

General

Target Model: Robot Kyle

Pose Interval: 0.1

Config Override (Optional): None (Motion Match Config Module)

Figure 2.2 'PreProcessor settings foldout'

Config Override - The configuration override property is an optional slot for placing a configuration module. Configuration models allow you to make a standalone asset that you can re-use between pre-processors and animation modules for convenience. See Section 2.1.4 for more detail on Configuration Override Modules.

2.1.3 Trajectory Configuration

In order for motion matching to work, it needs to match a predicted trajectory (past actual and future desired) to the trajectories of all animation poses (Figure 2.3). This is where you define how many of these points there are and what time they are in the past and future. I recommend 4-5 points for a good balance of quality and performance. However, MxM supports between 1-12 trajectory points.

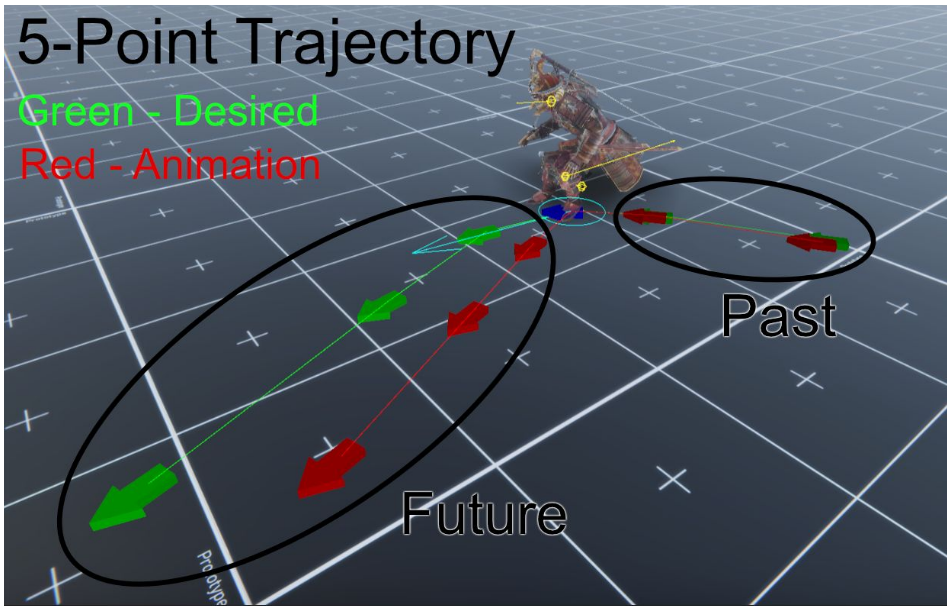
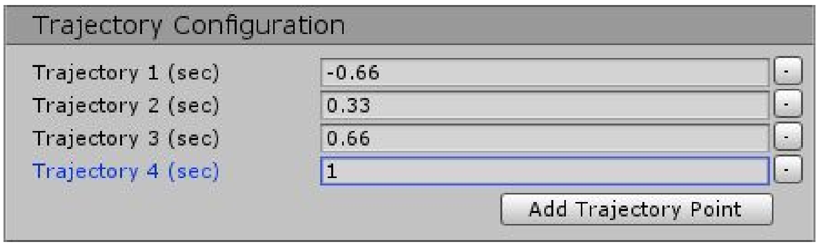


Figure 2.3 ‘Trajectory points shown at runtime’

For timing it is recommended to use values between -1s in the past and 1s in the future. Figure 2.4 shows a good example of trajectory configuration which has worked well in the past. For a 5 point trajectory, a second past trajectory point is recommended.



Trajectory Configuration

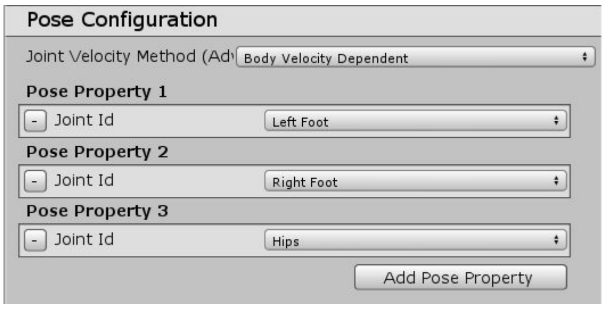
Trajectory 1 (sec)	-0.66
Trajectory 2 (sec)	0.33
Trajectory 3 (sec)	0.66
Trajectory 4 (sec)	1

Figure 2.4 ‘Typical trajectory configuration foldout’

Note that the inspector enforces progressively increasing numbers so your first trajectory point must be the lowest number.

2.1.4 Pose Configuration

The third foldout in the inspector contains the pose configuration. This is where you can choose which rig joints you want the system to match. MxM supports matching between 1-8 joints. However, it is recommended to only use 3 unless there is a specific reason, for example, matching a sword hand in a sword fighting game.


Pose Configuration

Joint Velocity Method (Adv: Body Velocity Dependent

Pose Property 1

Joint Id: Left Foot

Pose Property 2

Joint Id: Right Foot

Pose Property 3

Joint Id: Hips

Figure 2.5 ‘Pose configuration’

MxM will automatically detect if you are using a generic rig or a human rig. The joints showing up in the list will be displayed appropriately for the different rig types.

Don’t be tempted to match as many joints as possible. It is a common misconception that you need to match a lot of joints to achieve good results. This is completely false, three (3) joints is sufficient for general locomotion. This includes the left foot, right foot and a central joint like the hips, chest or neck. If you have sword fighting in your game it may be beneficial to also match the tip of the sword or sword hand, depending on how your game is designed.

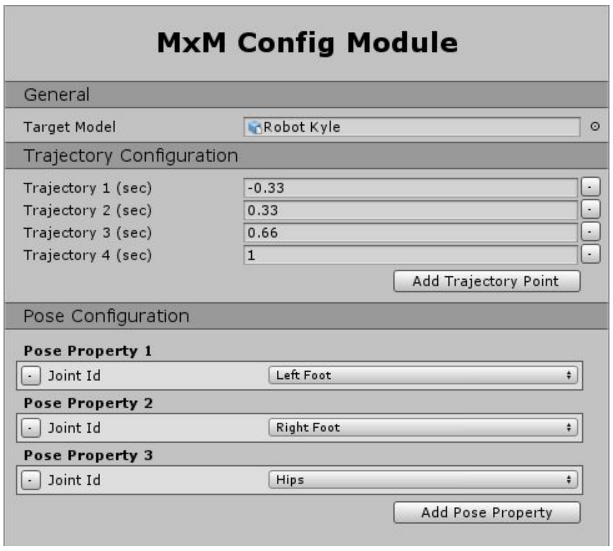
As a general rule of thumb, pick joints that drive motion and that move a lot. That is why the feet are so important. The hands are not as important as they may seem, as their motion is generally in-sync with the feet and vary more unpredictably. For a climbing locomotion, you would likely match the hands as they tend to drive the animation while climbing instead of the feet.

2.1.4 Configuration Overrides

Configuration overrides are a completely optional convenience feature that allows you to setup a configuration in a standalone asset file instead of re-doing it for every pre-processor you make.

To create a ‘Configuration Override Module’, right click in the project view and choose *‘Create / MxM / Core / Modules / Motion Matching Config’*

The inspector for the config module (Figure 2.6), is almost identical to the configuration section of the pre-processor, except it doesn’t specify pose interval.


MxM Config Module

General

Target Model: Robot Kyle

Trajectory Configuration

Trajectory 1 (sec): -0.33
Trajectory 2 (sec): 0.33
Trajectory 3 (sec): 0.66
Trajectory 4 (sec): 1

Pose Configuration

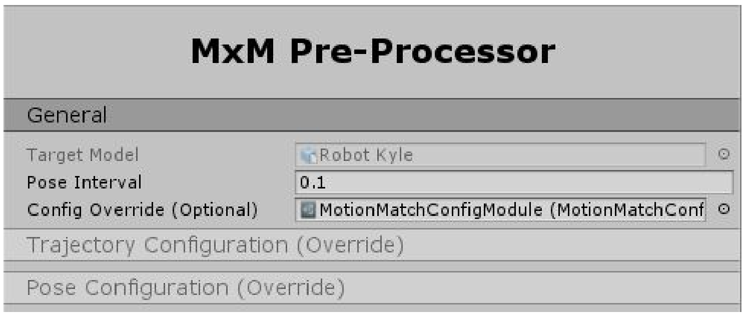
Pose Property 1
Joint Id: Left Foot

Pose Property 2
Joint Id: Right Foot

Pose Property 3
Joint Id: Hips

Figure 2.6 ‘MxM Config Module Inspector’’

When a configuration module asset is set in the ‘Config Override’ property of the general foldout of the MxM Pre-Processor, the overridden values in the pre-processor will be greyed out and un-editable as shown in Figure 2.7.


MxM Pre-Processor

General

Target Model: Robot Kyle
Pose Interval: 0.1
Config Override (Optional): MotionMatchConfigModule (MotionMatchConf

Trajectory Configuration (Override)

Pose Configuration (Override)

Figure 2.7 ‘MxM Pre-Processor with override configuration’’

2.1.5 Animations

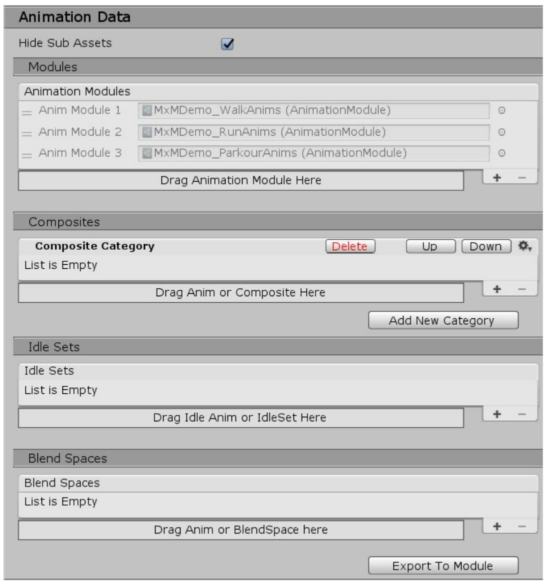
The fourth foldout in the inspector is where animations should be added to use with the character. Drag and drop animations into the ‘drag/drop’ box in the appropriate sections. There are four subsections here:

Modules - Add any animation modules here (see section 2.16 for details)

Animation Composites - Add all your ‘non-idle’ animations here.

Idle Sets - This is where you add idle animations (but not all of them! See the Idle Set Section 4 of the manual for more detail)

Blend Spaces - This is where you create blend spaces which work similarly to blend trees in mecanim. See Section 5 for details on blend spaces.


Animation Data

Hide Sub Assets ✓

Modules

Animation Modules

Anim Module 1 MxMDemo_WalkAnims (AnimationModule)

Anim Module 2 MxMDemo_RunAnims (AnimationModule)

Anim Module 3 MxMDemo_ParkourAnims (AnimationModule)

Composites

Composite Category

List is Empty

Idle Sets

Idle Sets

List is Empty

Blend Spaces

Blend Spaces

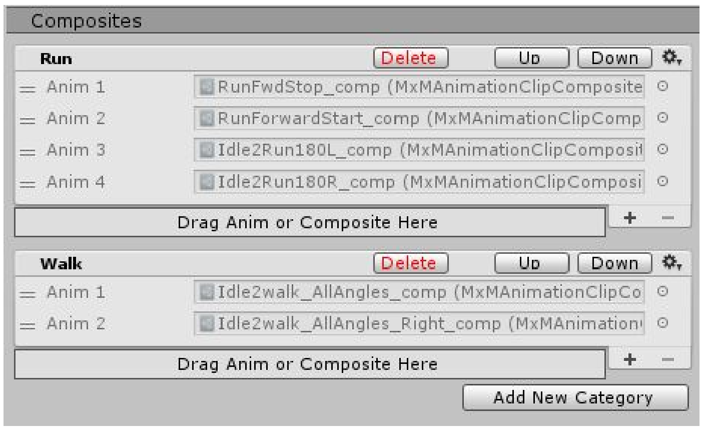
List is Empty

Drag Animation Module Here
Delete
Up
Down
Drag Anim or Composite Here
Add New Category
Drag Idle Anim or IdleSet Here
Drag Anim or BlendSpace here
Export To Module

Figure 2.8 ‘Pre-processor animations foldout’

As you add animations you will notice they will be added to a reorderable list. Double click any of these entries in the list to open their corresponding editors.

For composites, it is possible to create category sub lists. To do so, simply click on the ‘New Category’ button and a new list will appear. Lists can be re-named by clicking on their titles. Note that these category sub lists are cosmetic and for organizational purposes only. They do not affect the system in any way.


Composites

Run

Anim 1 RunFwdStop_comp (MxMAnimationClipComposite

Anim 2 RunForwardStart_comp (MxMAnimationClipComp

Anim 3 Idle2Run180L_comp (MxMAnimationClipComposit

Anim 4 Idle2Run180R_comp (MxMAnimationClipComposi

Walk

Anim 1 Idle2walk_AllAngles_comp (MxMAnimationClipCo

Anim 2 Idle2walk_AllAngles_Right_comp (MxMAnimation

Figure 2.9 ‘Multiple Categories’’

Double clicking on any of the object fields in these lists will open their respective editors. Composites, idle sets and blend spaces all have their own separate editors as shown in Figure 2.10. These editors will be explained in detail later in the manual and are vital to the quality of the animation if you are using cut clips.

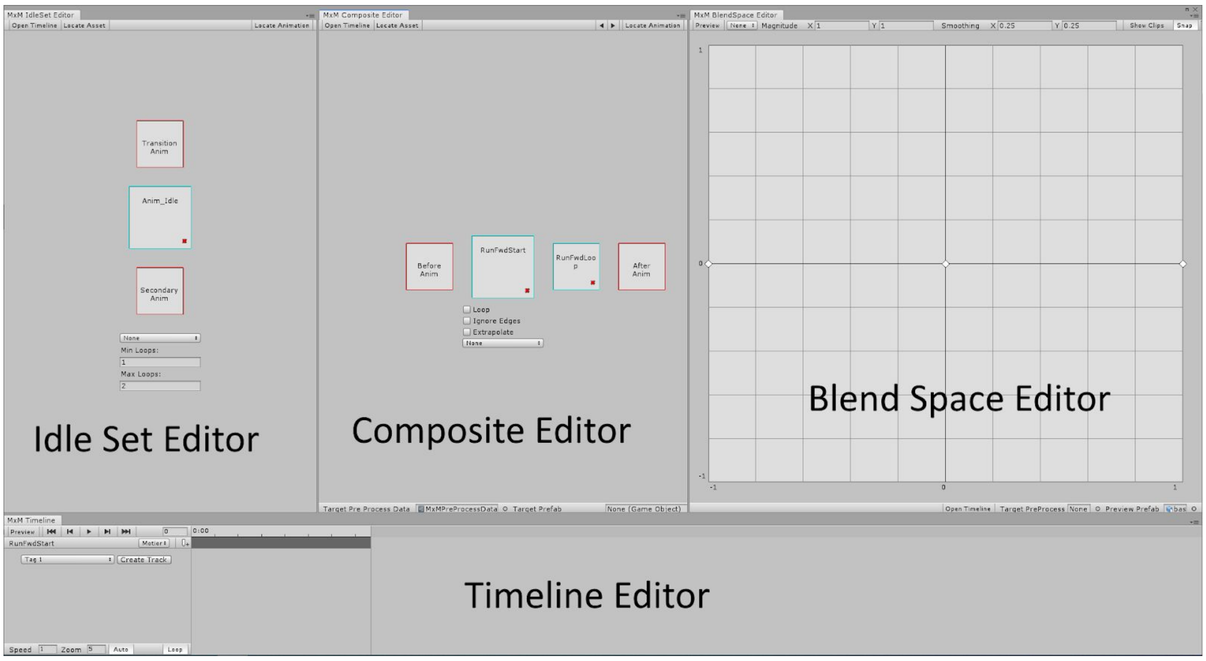


Figure 2.10 ‘MxM Animation Windows’

Hide Sub Assets - It is worth noting that for each composite, idle set and blend space, a new asset object is being created and added as a child to the PreProcessor asset. By default these sub-assets are hidden to avoid clogging up the project view. However, if you do need access to them, simply uncheck the ‘Hide Sub Assets’ toggle under ‘Animation Data’.

This can be useful because it is possible to drag and drop these sub assets directly into the composites, idle set or blend space sections of a different pre-processor to make a duplicate. However, it is best to leave them hidden most of the time.

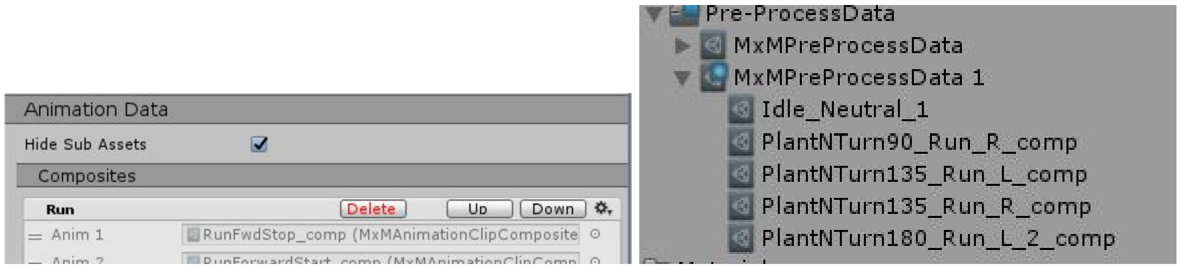
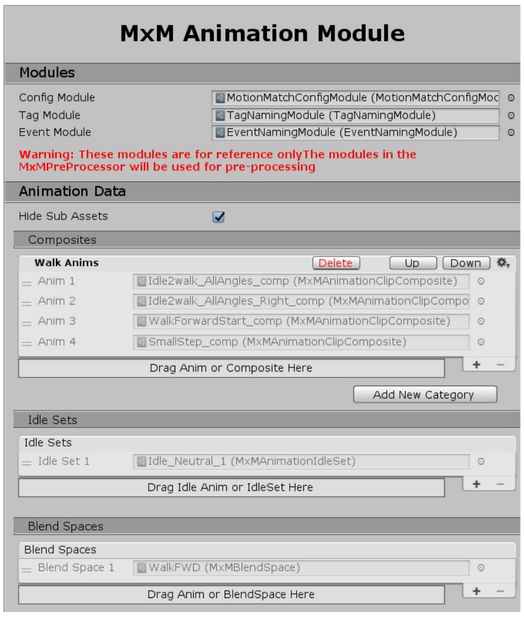


Figure 2.11 ‘Pre-processor sub assets’

2.1.6 Animation Modules

For a better workflow, it is recommended to use Animation modules instead of setting up your animations directly in the MxMAnimator. Animation modules are a way to separate your animation data from the Pre-Processor. This is very useful if you want to share animation data between pre-processors, or if you want to be able to add and remove animation data for testing and iteration without destroying that data.

Animation modules can be created by right clicking in the project view and choosing *'Create / MxM / Core / Modules / Animation'*. This will create a new asset with an inspector that closely resembles that of the Pre-Process Data animation foldout (Figure 2.12).


MxM Animation Module

Modules

Config Module: MotionMatchConfigModule (MotionMatchConfigMod
Tag Module: TagNamingModule (TagNamingModule)
Event Module: EventNamingModule (EventNamingModule)

Warning: These modules are for reference onlyThe modules in the MxMPreProcessor will be used for pre-processing

Animation Data

Hide Sub Assets: ✓

Composites

Walk Anims
Anim 1: Idle2walk_AllAngles_comp (MxMAnimationClipComposite)
Anim 2: Idle2walk_AllAngles_Right_comp (MxMAnimationClipCompo
Anim 3: WalkForwardStart_comp (MxMAnimationClipComposite)
Anim 4: SmallStep_comp (MxMAnimationClipComposite)

Idle Sets

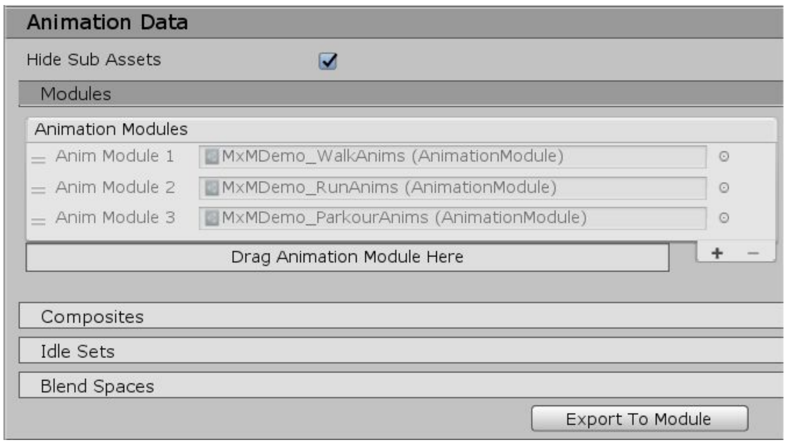
Idle Sets
Idle Set 1: Idle_Neutral_1 (MxMAnimationIdleSet)

Blend Spaces

Blend Spaces
Blend Space 1: WalkFWD (MxMBlendSpace)
Delete
Up
Down
Drag Anim or Composite Here
Add New Category
Drag Idle Anim or IdleSet Here
Drag Anim or BlendSpace Here

Figure 2.12 'Animation Module Inspector'

Animation modules can then be easily added into the Modules foldout of the 'Animation Data' section of your pre-processor by dragging them into the box as shown in Figure 2.13.


Animation Data

Hide Sub Assets: ✓

Modules

Animation Modules
Anim Module 1: MxMDemo_WalkAnims (AnimationModule)
Anim Module 2: MxMDemo_RunAnims (AnimationModule)
Anim Module 3: MxMDemo_ParkourAnims (AnimationModule)

Figure 2.13 'Animation Module Inspector'

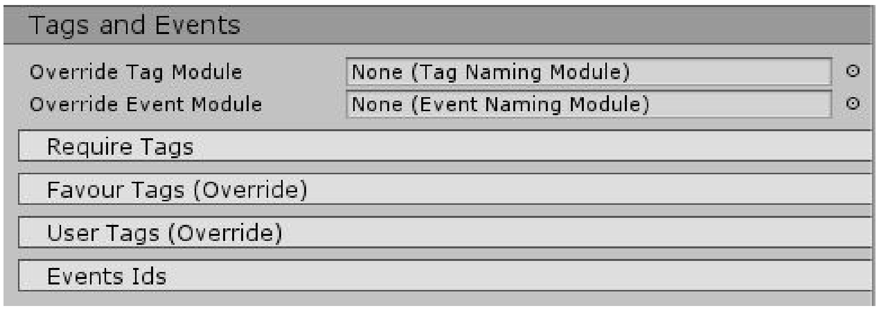
When using animation modules, it is vital to use a complete modular workflow, utilising the 'Motion Matching Config Module' (Section 2.1.4), 'Tag Naming Module' (Section 2.1.7) and 'Event Naming Module' (Section 2.1.7). These modules should be referenced in the Modules foldout of the 'Animation Module' inspector as well as in the preprocessors you intend to use the module for. This allows consistent configuration, tag naming and event naming between modules and the pre-processors.

Exporting Animation Modules from Pre-Processors

If you already have a pre-processor with animation data and would like to convert to a modular workflow, you can export that animation data to a module by clicking the ‘Export To Module’ button at the bottom right of the AnimationData foldout (Figure 2.13).

2.1.7 Tag and Event Names

The ‘Tags and Events’ foldout has four sub foldouts. This is where you define what tags, favour tags, user tags and event ids exist so you can use them to markup your animations and call for them during runtime.


Tags and Events

Override Tag Module	None (Tag Naming Module)
Override Event Module	None (Event Naming Module)

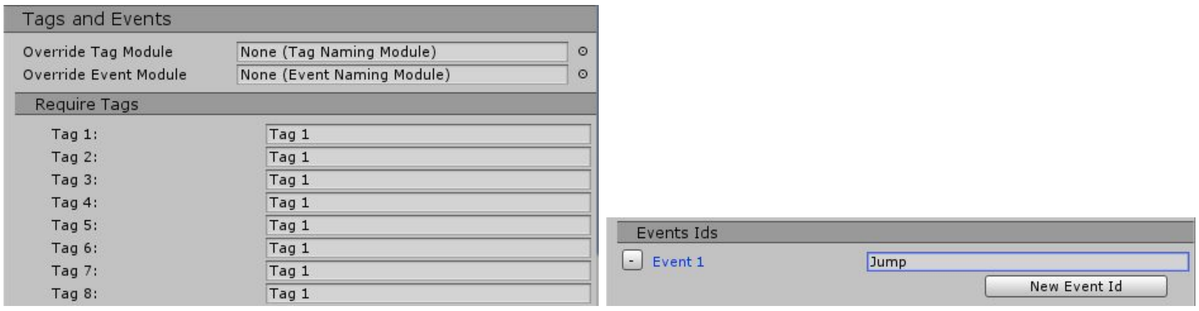
Figure 2.14 ‘Pre-processor Tags and Events Foldout’

Require Tags - Using the tagging editor (See section 3.3), you can mark up animation data to allow more control over what kind of animations you want playing. For example, you may want your character to move differently with their sword drawn than when it is sheathed. In this case you would probably create a required tag called “Combat Stance” and apply that tag to all animations that use that stance. There are 30 customisable ‘require tag’ slots.

Favour Tags - Favour tags are similar to normal tags. However, they are not exclusive, meaning a pose does not have to contain the currently favoured tag to be a potential candidate. If a pose has the same favour tags set as the currently favoured tags then it will just be more likely to be chosen. The amount of favour can be set at runtime by the user. An example use case for this is to favour tagging walking and running separately to fine tune when running only animations play and walking only animations play at runtime. This is generally only necessary if the animation isn’t behaving.

User Tags - User tags are as the name describes. They are tags that are defined by the user. On their own they do not do anything. However, it is possible for the user to query these tags at runtime to run custom logic based on the active tags.

Event Ids - Events are used to trigger action animations that you only want to play in certain circumstances. For example, you only want the character to swing their sword when the player presses the attack button. For now we are just defining what events exist and this can be revisited at any time. Events are also marked on animations in the timeline editor which will be covered in Section 3.3.


Tags and Events

Override Tag Module: None (Tag Naming Module)
Override Event Module: None (Event Naming Module)

Require Tags

Tag 1: Tag 1
Tag 2: Tag 1
Tag 3: Tag 1
Tag 4: Tag 1
Tag 5: Tag 1
Tag 6: Tag 1
Tag 7: Tag 1
Tag 8: Tag 1

Events Ids

- Event 1: Jump

New Event Id

Figure 2.15 'Tags & Events foldout'

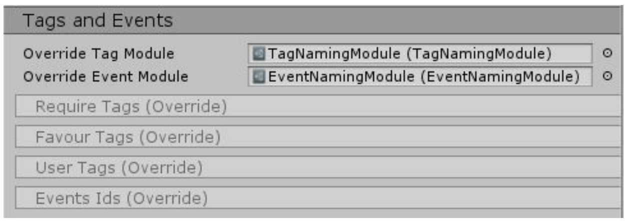
Note: You don't need an event ID for every single action animation, multiple similar actions can have the same event ID and MxM will pick the appropriate one based on the gameplay metrics you provide it as well as motion matching. Events are explained in detail in Section 3.3.1.

Override Modules - The override module properties displayed at the top of the 'Tag and Events' foldout can be used to override all defined tags and event Ids in the pre-processor with a standalone asset. This is a convenience feature to help avoid rewriting tag and event definition for every pre-processor. It works in the same way as with override configuration modules explained in Section 2.1.4.

It is highly recommended to use tag and event naming modules for a better workflow, particularly if you are also using animation modules. Tag and event naming modules make it easier to have consistent tags and events between all your modules.

To create an override tag or event module, right click in the project view and choose 'Create / MxM / Core / Modules / TagNaming (or EventNaming). The inspectors for these standalone module assets are intended to mirror the corresponding sections of the MxMPreProcessor inspector.

When these override modules are slotted in the Pre-Processor, the tag and event sub foldouts will be greyed out and un-editable as shown in Figure 2.16


Tags and Events

Override Tag Module: TagNamingModule (TagNamingModule)
Override Event Module: EventNamingModule (EventNamingModule)

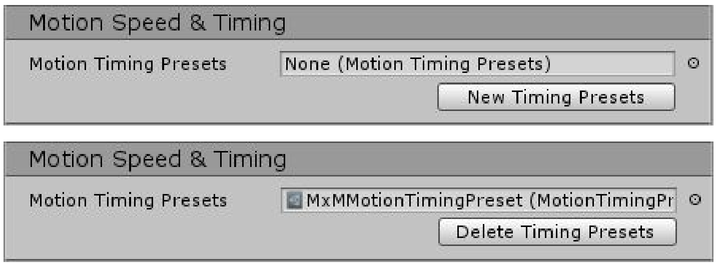
Require Tags (Override)
Favour Tags (Override)
User Tags (Override)
Events Ids (Override)

Figure 2.16 'Tags & Events foldout'

2.1.8 Motion Timing Presets

The next foldout contains the motion timing presents. One of the biggest downfalls of mocap data is that it is relatively slow compared to motion in games. Actors can try their best to perform fast and responsive movements but it's rarely fast enough. As a result, MxM has tools to modify the speed of 'parts of animations' to match gameplay. The motion timing presets are basically a blackboard of preset motion timing and speed variables that can be accessed in the motion timing editor (Section 7.2) across all animations in a pre-processor.

These presets are not compulsory so it can be left blank. However, to create a preset asset, simply click the “Create Motion Timing Presets” button. This will generate an asset as a child of the preprocessor for use later.


Motion Speed & Timing
Motion Timing Presets: None (Motion Timing Presets)
New Timing Presets

Motion Speed & Timing
Motion Timing Presets: MxMMotionTimingPreset (MotionTimingPr
Delete Timing Presets

Figure 2.17 ‘Motion timing presets foldout, Top - No presets, Bottom - Presets added’

<u>Note:</u> This is more of a prototyping tool. The animations it generates are in text format and therefore, not typically suitable for production.

2.1.9 Pre-Process

The final foldout in the PreProcessData inspector simply contains a button to pre-process your animation data and a setting to configure the pre-processing phase. The pre-process settings are as follows:

- **Generate modified clips** - if you have made speed modifications to animation clip composites (Section 7), checking this option will ensure that these clips get generated before pre-processing if they haven’t already.

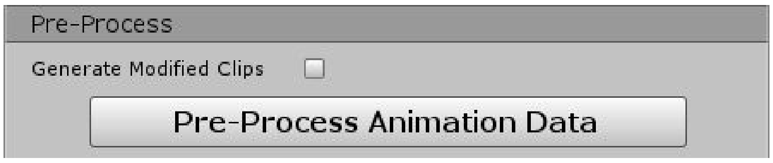

Pre-Process
Generate Modified Clips ☐
Pre-Process Animation Data

Figure 2.18 ‘Pre-process foldout’

After clicking the ‘Pre-Process Animation Data’ button you will be prompted to specify a file directory to save the resulting animation data. Select an appropriate directory and filename, and choose save. The pre-processor may take some time to run especially if new clips need to be generated.

Once complete, the new ‘MxMAnimData’ asset will be generated and placed in the directory that you specified. If you chose to generate speed modified clips, a folder will be created in the same directory with the same name as the ‘MxMAnimData’ and your newly generated clips will be placed in there.

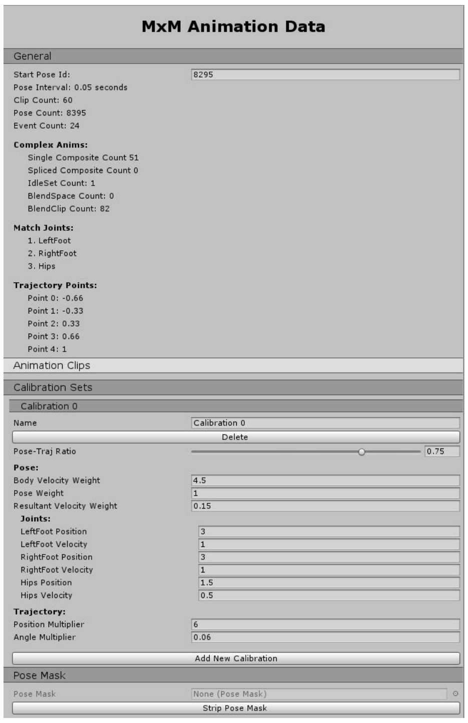
The generated MxMAnimData contains all the pose information that MxM needs to perform motion matching at runtime. The preprocessor asset itself does not need to ship with your game but it should be kept during development for re-processing in case you want to make changes later (and you will).

Note: Overwriting an existing MxMAnimData asset will ensure that the existing configuration (motion match weightings) persist even after overwriting.

Note: Pre-processing should be re-done every time MxM is updated.

2.2 The Animation Data (MxMAnimData)

The result of every successful pre-process phase is an ‘MxMAnimData’ asset. This is mostly pure data that should not be edited. However, you can view meta-data for the asset in the inspector. You can also see all the animations that the MxMAnimData uses but this is not editable.



```
MxM Animation Data

General
Start Pose Id:                    8295
Pose Interval: 0.05 seconds
Clip Count: 60
Pose Count: 8395
Event Count: 24

Complex Anims:
    Single Composite Count 51
    Spliced Composite Count 0
    IdleSet Count: 1
    BlendSpace Count: 0
    BlendClip Count: 82

Match Joints:
    1. LeftFoot
    2. RightFoot
    3. Hips

Trajectory Points:
    Point 0: -0.66
    Point 1: -0.33
    Point 2: 0.33
    Point 3: 0.66
    Point 4: 1
Animation Clips
Calibration Sets
  Calibration 0
Name                              Calibration 0
                     Delete
Pose-Traj Ratio                                   0.75
Pose:
Body Velocity Weight              4.5
Pose Weight                       1
Resultant Velocity Weight         0.15
  Joints:
  LeftFoot Position                3
  LeftFoot Velocity                1
  RightFoot Position               3
  RightFoot Velocity               1
  Hips Position                    1.5
  Hips Velocity                    0.5
Trajectory:
Position Multiplier               6
Angle Multiplier                  0.06

                Add New Calibration
Pose Mask
Pose Mask                         None (Pose Mask)
                  Strip Pose Mask
```

Figure 2.19 ‘MxMAnimData metadata inspector’

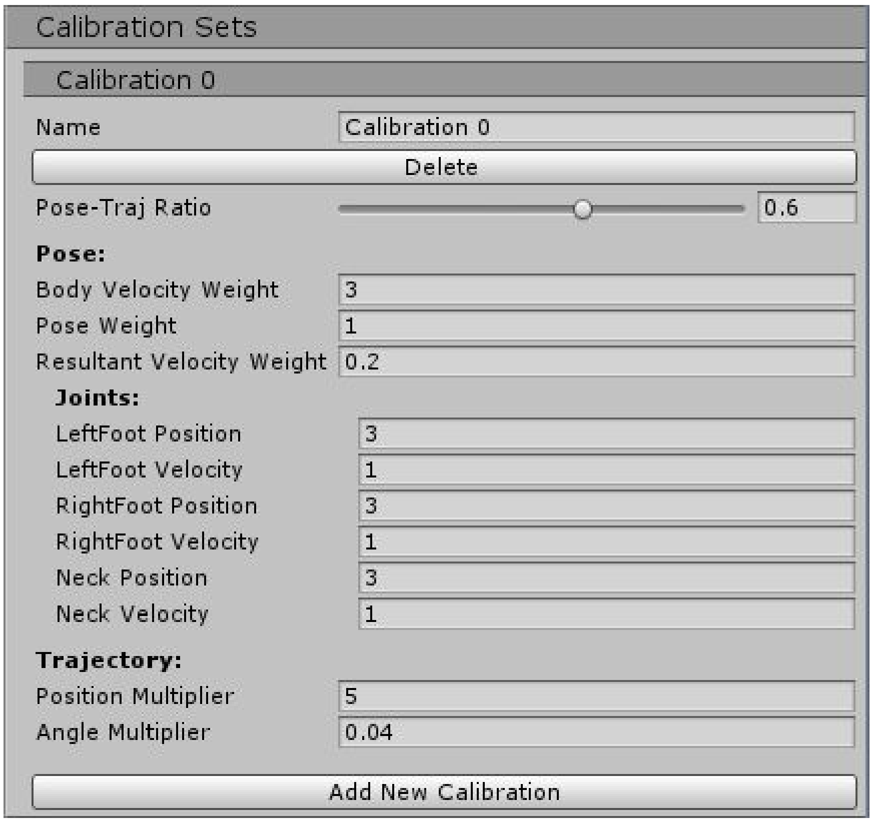
2.2.1 Calibration Sets

The MxMAnimData inspector does allow creating and editing of calibration sets. Calibration sets include weightings and parameters to tell the system how important certain properties are to match. Most of the time the default calibration settings are sufficient but can be modified at any point. Weightings can be specified for the following:

● Body Velocity	[Default: 3]
● Overall Pose weight	[Default: 1]
● Resultant Velocity weight	[Default: 0.2]
● Joint Position (each joint specified in the pre-process data)	[Default: 3]

● Joint Velocity (each joint specified in the pre-process data)	[Default: 1]
● Trajectory Position	[Default: 5]
● Trajectory Facing Direction	[Default: 0.04]

Calibration will be discussed in detail through section 6. However, the most important setting in calibration is the 'pose-trajectory ratio' slider. The farther to the right this slider is, the more responsive the animation and farther to the left, the more fluid the animation will be. Most of the time you will keep this somewhere in the middle but it can be tweaked at any time. Motion matching is a balancing act between fluidity (pose) and responsiveness (trajectory).

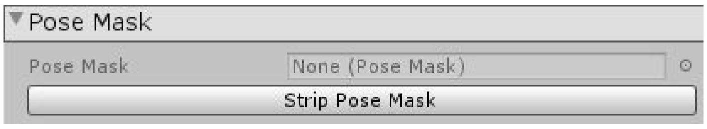


Name: Calibration 0
Pose-Traj Ratio: 0.6
Pose:
Body Velocity Weight: 3
Pose Weight: 1
Resultant Velocity Weight: 0.2
Joints:
LeftFoot Position: 3
LeftFoot Velocity: 1
RightFoot Position: 3
RightFoot Velocity: 1
Neck Position: 3
Neck Velocity: 1
Trajectory:
Position Multiplier: 5
Angle Multiplier: 0.04

Figure 2.20 'Calibration foldout'

2.2.2 Pose Mask

A pose mask is a data object which contains information on what animation poses the system actually uses. Each pose mask relates to an animation data and when created it will be embedded in the animation data. It can be stripped out here by clicking the 'Strip Pose Mask' button.



Pose Mask: None (Pose Mask)

Figure 2.21 'Pose mask foldout'

More details on pose masks in Section 12 and 13.5.

2.3 The Motion Matching Animator (MxMAnimator)

The MxMAnimator is the runtime component (monobehaviour) that needs to be placed on your character model. It runs the motion matching on your character similar to how the standard Animator component works with mecanim.

2.3.1 Required Components

The MxMAnimator doesn't work alone. It requires some other components to collaborate with. These components are as follows:

Animator (Mandatory) - An animator component is required for MxM to interface with Unity's underlying animation system. It does not require an Animator Controller to be attached, unless you want to create a hybrid system as explained in Section 8.2.

IMxMTrajectory (Mandatory) - A component that implements the IMxMTrajectory interface is required. The purpose of this component is to generate a predicted trajectory (past and future) which matches the trajectory configuration set in the MxMPreProcessor. MxM will automatically extract data from the trajectory generator at runtime to drive animation.

For most cases, the *MxMTrajectoryGenerator* component, that ships with MxM, is sufficient. It automatically records the past trajectory and predicts future trajectory based on user input. MxM also ships with a simple NavMesh based trajectory generator for AI. However, you may need to create your own custom TrajectoryGenerator to suit specific gameplay needs.

The MxMTrajectoryGenerator inherits from MxMTrajectoryGeneratorBase (an abstract IMxMTrajectory implementation) which is a useful intermediate component for recording of the past and managing the containers for your future trajectory. All that is required is to fill the future trajectory containers based on your gameplay.

Section 10.1 explains how to use the MxMTrajectoryGeneratorBase as a starting point for your own motion matching gameplay controller.

IMxMRootMotion (Non-Mandatory) - If you choose to handle root motion with custom code, you will need to have a component attached that implements the IMxMRootMotion interface. The MxMAnimator can be set to send root motion data (along with warping) to the IMxMRootMotion interface for manual handling (Section 2.3.2).

Additionally, when root motion is turned on, MxM applies it directly to the transform by default. If you are using a character controller that requires a specific 'Move' function to operate, then you will need to apply the root motion to the controller with your custom IMxMRootMotion component.

MxM ships with an 'MxMRootMotionApplicator' component which implements IMxMRootMotion. Its purpose is to apply gravity to the root motion and convey the resulting root motion to a character controller via a 'GenericControllerWrapper' component. MxM comes with a wrapper made for the default Unity 'Character Controller' component. However, you can make your own wrapper for other controllers by inheriting from 'GenericControllerWrapper' or you can make your own custom IMxMRootMotion component that interfaces directly with your character controller of choice.

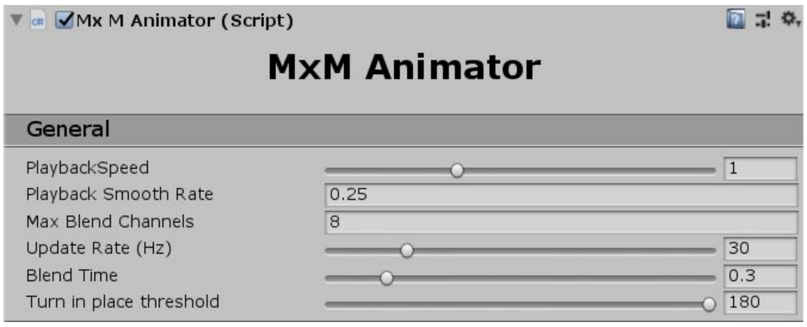
This setup has been designed to allow maximum flexibility so that any character controller can be used. An integration for Kinematic Character Controller is available for download on the Discord channel.

Note: Root motion can be turned off completely and is not required at runtime for MxM to work.

Note: When referring to character controller, this guide is referring to a capsule that enables movement with basic functionality, not a fully implemented gameplay character controller like 'OOTI' or Unity Standard Third Person controller.

2.3.2 General

The first fold out allows you to set some basic settings to configure the runtime MxMAnimator. The following settings can be configured as you desire but be aware of the recommended defaults.



PlaybackSpeed	1
Playback Smooth Rate	0.25
Max Blend Channels	8
Update Rate (Hz)	30
Blend Time	0.3
Turn in place threshold	180

Figure 2.22 ' MxMAnimator General Foldout'

Playback Speed - The playback speed of animation generated by MxM. Default is set to 1 for normal animation speed. Note that trajectory longitudinal warping will affect this multiplier at runtime if you choose to use it.

Playback Smooth Rate - When the playback speed is changed through code by setting the '*MxMAnimator.DesiredPlaybackSpeed'* property, the playback speed will smoothly change to the rate specified rather than instantly snapping.

Max Blend Channels - The maximum number of blend channels that MxM will use to blend in and out animations. Keeping this low will improve performance. However, having it too low may result in animation jumping.

Update Rate (Hz) - This is the rate at which animator 'matching' updates should run. It is recommended to set this value at 30Hz for most cases. The update rate can be increased to improve performance and it can even be used as a method of motion matching LOD, increasing it for characters further away and decreasing it for characters that are close. For best responsiveness, keep this value low but avoid updating every frame. 30 - 60Hz is sufficient.

Note: The update rate is not the frame rate. MxM still updates some logic every frame. The update rate specified by this value is how frequently the motion matching cost function is run to pick a new pose to jump to. In between these searches, animation plays and blends as normal.

Blend Time - The time it takes for an animation to fully blend in once chosen. Values between 0.2 - 0.4s are recommended. Higher blend rates provide more smooth transitions between animations but also reduce responsiveness and may muddy the animation and increase foot sliding. Lower values provide better responsiveness but less fluid animation. With highly dense mocap the blend time can be set low as blending will be minimal.

Turn In Place Threshold - MxM Automatically detects when the character is in an idle state. However, depending on the turn in place animations you have available, it may be beneficial to set a Turn in Place Threshold so that turn in place only occurs after the angle difference is beyond this threshold. As a rule of thumb start with this set to your narrowest turn in place animation.

2.3.3 Animation Data Foldout

This is where MxMAnimData assets, generated in the preprocessing stage, should be placed in the MxMAnimator. While there is no AnimData in the list you will notice a bold red warning and the rest of the inspector will be greyed out.

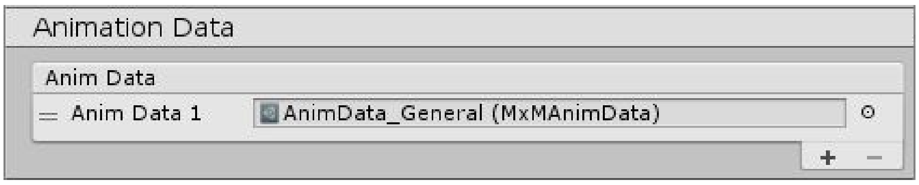
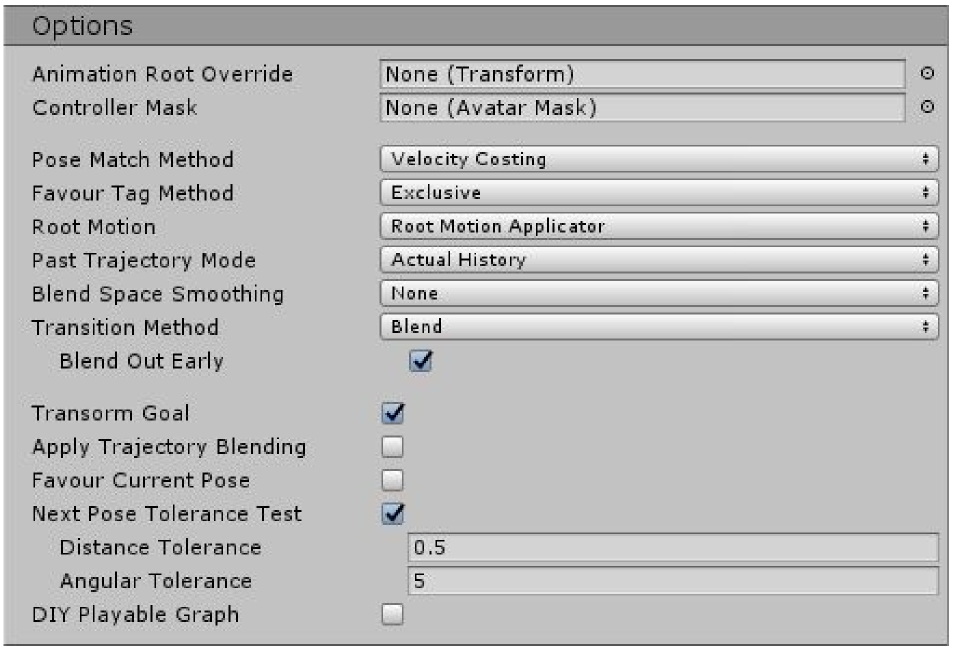


Figure 2.23 ' MxMAnimator Animation Data Foldout'

Multiple animation data assets can be added and switched between at runtime but this is not recommended unless there is a distinct transition between styles where you need different animations with different matching configurations (i.e. from running to climbing needs to match the hands instead of the feet).

WARNING: Switching between AnimData in the list is done manually through code. MxM will not search multiple AnimData at one time. This is for technical and performance reasons.

2.3.4 Options


Options

Animation Root Override	None (Transform)
Controller Mask	None (Avatar Mask)
Pose Match Method	Velocity Costing
Favour Tag Method	Exclusive
Root Motion	Root Motion Applicator
Past Trajectory Mode	Actual History
Blend Space Smoothing	None
Transition Method	Blend
Blend Out Early	✓
Transorm Goal	✓
Apply Trajectory Blending	☐
Favour Current Pose	☐
Next Pose Tolerance Test	✓
Distance Tolerance	0.5
Angular Tolerance	5
DIY Playable Graph	☐

Figure 2.24 ‘MxMAnimator Options Foldout’

Animation Root Override - By default MxM uses and applies movement to the transform that the component is attached to. However, if for some reason this is not desirable, the root transform can be overridden with another transform (e.g. a parent object) by slotting it in this field. Leave blank by default and only use it if necessary.

Controller Mask - This property allows you to specify an avatar mask to use for the mecanim ‘Animator Controller’, if you decide to use one in hybrid with MxM. This can allow the use of mecanim to control upper body layers if desired.

Pose Match Method - The pose match method is the algorithm used by MxM to run the cost function under the hood. The default is ‘Velocity Costing’ and for most cases this should be left alone. The ‘Basic’ method can be used for a very slight increase in performance for cost of quality. Consider using ‘Basic’ as a method of LOD for far away characters.

Favour Tag Method - This setting defines how MxM will handle favour tagging at runtime. There are three options:

- Exclusive - Poses must have all requested favour tags to be given favour
- Inclusive - Poses must have at least one requested favour tag to be given favour
- Stacking - Poses will be given favour exponentially for the number requested favour tags that it matches.

Root Motion - This setting is used to describe how root motion should be handled. It can have the following options:

- *On* - Root motion applied directly to the transform (or override animation root)
- *Off* - root motion will not be applied (and neither animation warping)

- *RootMotionApplicator* - Root Motion will be applied through a root motion applicator
- *RootMotionApplicator_AngularErrorWarpingOnly* - Root motion will not be applied. However, angular error warping will be applied through a root motion applicator.

For more details on custom handling of root motion, please see section 10.3.

Past Trajectory Mode - This setting is used to tell MxM what method to use when determining past trajectory. Settings are as follows:

- *'Actual History'* - the past trajectory will be recorded at runtime in world space. This is the default setting and should be used for most cases
- *'Copy from last pose'* - This copies the past trajectory from the last pose. This can be very useful in situations where there is going to be a lot of animation warping and speed manipulation, or when you have lacking coverage in your animations (cut clips). This is the recommended setting for strafing when not using mocap data.

Blend Space Smoothing - MxM can match blend spaces as well as individual clips. Changes in the blend space position are generally smoothed via a specific method which is defined here. The smoothing methods work as follows

- None - no smoothing
- Lerp - The blend space position is lerped based on just the X smooth rate value
- Lerp2D - the blend space position is lerped in a 2D fashion based on both the X and Y smooth rate value
- Unique - The blend space position is lerped based on the unique X smoothing rates specified within the blend space itself.
- Unique2D - the blend space position is lerped based on the unique X and Y smoothing rates specified within the blend space itself.

Smooth Rate (Blend Space) - The 2D blend space smooth rate to be used with Lerp and Lerp2D blend space smoothing methods.

Transition Method - This popup defines how animations will be blended when jumping from one pose to another. 'None' results in no blending occuring, while 'Blend' crossfades the animations in a traditional manner. An additional method (inertial blending) may be added in the future but it is still in development.

Blend Out Early - This checkbox should only be set if you are using long mocap takes or you are heavily using runtime splicing for your transition animations. Results will vary depending on animation sets. Try this setting on and off and see what settings gives the best results.

Transform Goal - It is common to generate the goal (desired trajectory) at the origin of the world, isolated from the character's position and orientation. The example MxMTrajectoryGenerator does this. Check this toggle if you want MxM to make your trajectory relative to its own position. If you are already doing this, leave it unchecked.

Apply Trajectory Blending - Human motion is incredibly complex to simulate. Trajectory blending, blends your desired trajectory to that of the chosen pose to give a more realistic trajectory. It does this in a diminishing way to that earlier trajectory points are blended closer to the current pose with the last trajectory point having no trajectory blending whatsoever.

Trajectory Blend Weight - This value allows a weighting to be set for trajectory blending. It should be set to a value between 0 and 1. A higher value gives a more realistic trajectory but is also less responsive. While a lower value is more responsive but may have less fluid results. A value of 0.5 is a good starting point.

Favour Current Pose - This is a technique to help make animation smoother and reduce noise. The idea is that it should cost more to switch animation clips. This results in less erratic jumping between clips. However, it can also reduce responsiveness if the favour factor (below) is too low.

Pose Favour Factor - A multiplier to favour the current pose. This should be set between 0-1. A value of about 0.95 is recommended but 0.9 may be appropriate depending on your animations.

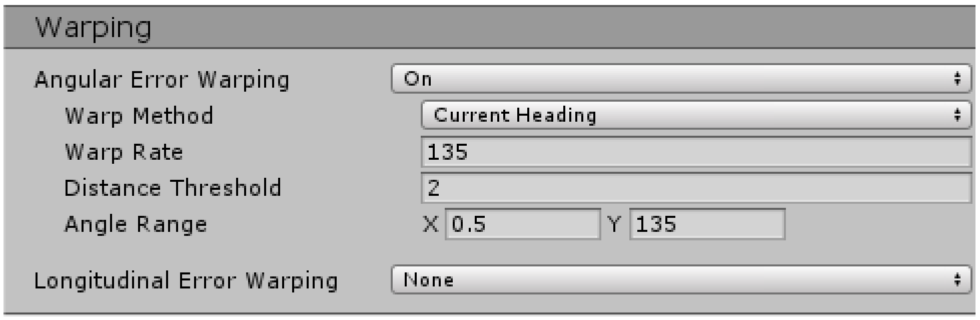
Next Pose Tolerance Test - This option enables and optimisation technique which does a test to see if the next pose trajectory in the current animation is a ‘close enough’ match to the desired trajectory. If it is, then the motion matching search is abandoned for that update and the current animation keeps playing. This provides significant performance increase with continuous actions like running straight and its impact is felt increasingly with more characters.

Distance Threshold - This is the distance threshold used in the ‘Next Pose Tolerance Test’ to see if a trajectory position is tolerable. If any trajectory point in the next pose is farther than this amount from the desired trajectory, a motion matching search will take place.

Angular Threshold - This is the angular threshold used in the ‘Next Pose Tolerance Test’ to see if a trajectory rotation is tolerable. If the angle difference between trajectory points in the next pose trajectory and desired trajectory is beyond this threshold, a motion matching search will take place.

DIY Playable Graph - This is a setting for advanced users only. This allows the user to create their own custom playable graph and then plug MxM into that playable graph as they desire. For more details see Section 11. Leave this setting off unless you are making a custom animation playable graph.

2.3.5 Warping


Warping

Angular Error Warping: On
Warp Method: Current Heading
Warp Rate: 135
Distance Threshold: 2
Angle Range: X 0.5 Y 135

Longitudinal Error Warping: None

Figure 2.25 ‘MxMAnimator Options Foldout’

Angular Error Warping - This is a technique used to compensate for gaps in animation coverage by warping the root rotation. For example, what if the player chooses to move at a 22.5 degree angle but you don’t have an animation for that angle. This is a trajectory error and procedural warping can be used to rectify it. There are three different modes for trajectory error warping:

- Off - Angular error warping is off
- On - The angular error will be warped over time by rotating the character root to compensate for the error.
- External - MxM will not call warping functions, rather it is up to the user to all these functions externally to MxM. This setting should only be used if the procedural error warping should only occur at specific times in the update loop (e.g. after a fixed update character controller moves)

Warp Method (Angular Error) - This is the method by which angular error warping will be calculated. The different methods include:

- Current Heading - Warp based on the angle between the characters current forward facing vector and the last point in the desired trajectory. (Not recommended for most cases)
- Trajectory Heading - Warp based on the angle between the last trajectory point of the current animation and desired trajectory. (Used for general locomotion)
- Trajectory Facing - Warp based on the angle between the characters current forward facing vector and the average facing vector of the trajectory (Used for strafe locomotion)

Warp Rate (Angular Error) - How fast angular errors are compensated for in degrees / second.

Distance Threshold (Angular Error) - How big the trajectory has to be before angular error warping will kick in. You generally don’t want angular error warping if you are standing still because the character will rotate on the spot.

Angle Range (Angular Error) - If the angular error is within this range, angular error warping will take place, otherwise it will be ignored, even if warping is on.

Longitudinal Error Warping - longitudinal error warping can be used to warp animation speed to compensate for errors in trajectory length. Currently there is only one method of longitudinal warping. Note also that longitudinal error warping only works if the MxMAnimator is told what scale to warp at by setting the property 'LongErrorWarpScale'. This is because there are multiple methods by which warping can be calculated. The MxMSimpleController has this built in.

- None - Longitudinal Error Warping will not occur
- Speed - Longitudinal errors will be warped by changing animation speed
- Stride - Achieve longitudinal error warping through integration with the 'Strider' asset (not included with MxM)

Speed Warp Limits (Long Error Warping) - It is not always a good idea to speed up or slow down animations beyond 10 - 20% otherwise animation fidelity is lost. This setting allows for settings the lower and upper limit of animation playback speed resulting from longitudinal error warping. Values between 0.8 (80% speed) to 1.2 (120% speed) are the maximum recommended limits for most animations.

2.3.6 Debug

The debug foldout of the MxMAnimator shows a number of debug options, including in-scene gizmo drawers as well as other tools. Each option is detailed below:

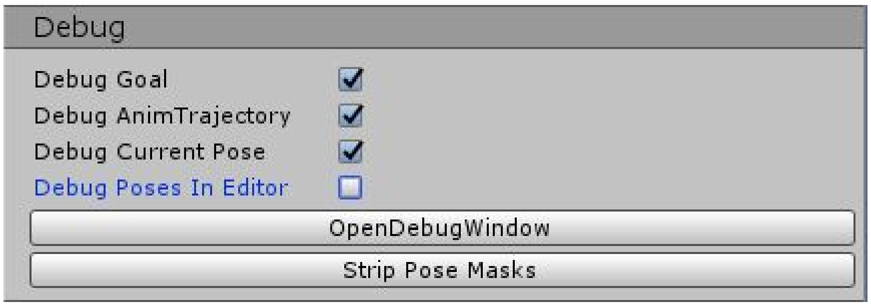

Debug
Debug Goal ✓
Debug AnimTrajectory ✓
Debug Current Pose ✓
Debug Poses In Editor ☐

Figure 2.26 'MxMAnimator debug foldout in edit mode'

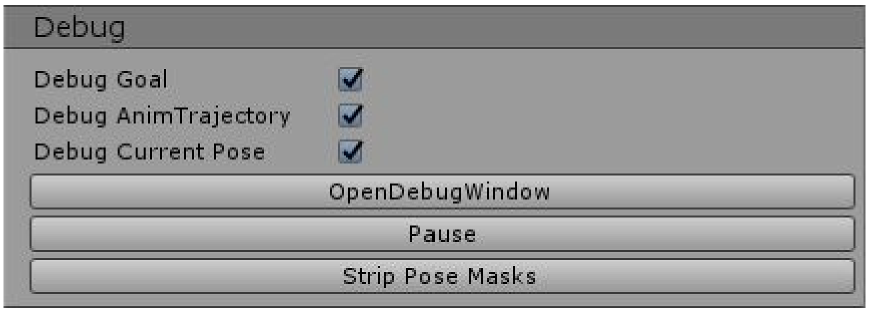

Debug
Debug Goal ✓
Debug AnimTrajectory ✓
Debug Current Pose ✓

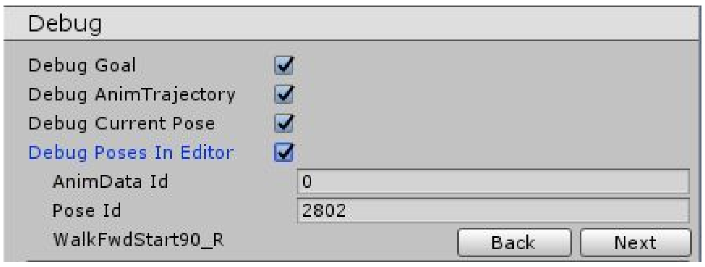
Figure 2.27 'MxMAnimator debug foldout in play mode'

Debug Goal - Check this to view the goal (Desired) trajectory during runtime. The goal will be indicated by green spheres with lines between them.

Debug Anim Trajectory - Check this to view the current animations trajectory during runtime. The current animation trajectory is indicated by red spheres with lines between them.

Debug Current Pose - Check this to visualise the position of matched joints as well as vectors for their velocities. These gizmos will show up on the character as yellow gizmos.

Debug Poses In Editor - Checking this toggle will add a character to the scene and allow you to view all the poses stored in the anim data including their trajectories and pose data. It is a good visualisation to check that your animations were processed properly.



Debug

Debug Goal ✓
Debug AnimTrajectory ✓
Debug Current Pose ✓
Debug Poses In Editor ✓
AnimData Id 0
Pose Id 2802
WalkFwdStart90_R

Figure 2.28 'Debug pose in editor options'

Open Debug Window - This button will open a debugging window for your character. This debugging window will only operate in editor at runtime. For more details on the Debug window see section 12.

Strip Pose mask - This will remove the pose masks currently attached to the MxMAnimData used for this animator.

Pause / UnPause - Will pause or unpause any animation output from the MxMAnimator

2.3.7 Callbacks

MxM has several callbacks which can be used to apply custom logic at certain times. These callbacks are displayed like any other Unity inspector callback as shown in Figure 2.28.

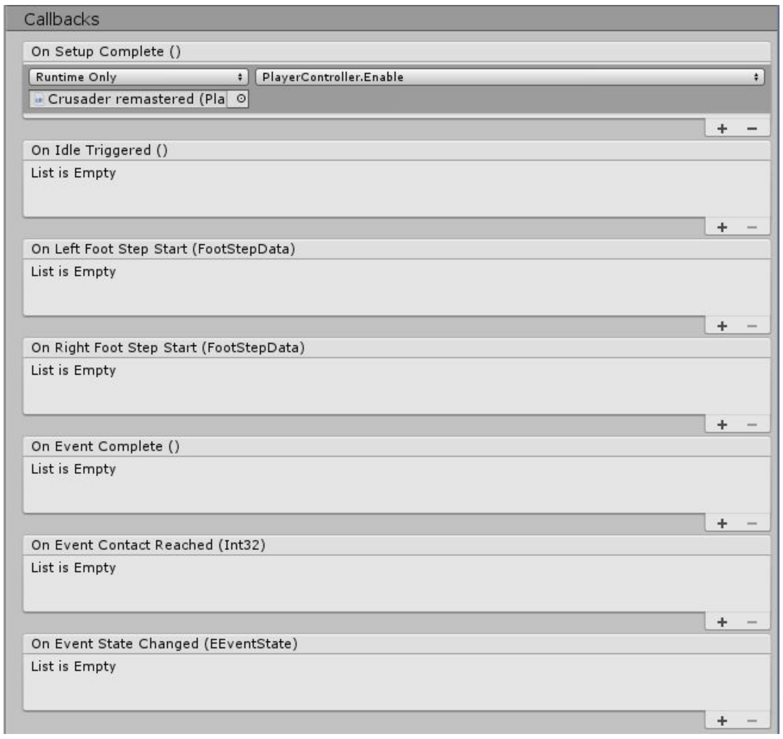


Figure 2.29 'MxMAnimator callbacks foldout'

The callbacks shown in Figure 2.29 are as follows:

- **OnSetupComplete** - this is called only once after the MxMAnimator is completely setup and ready to animate. Use this callback to trigger any logic that can only occur after MxM is ready to animate. One common application is to only enable a 'Rig Builder' component from Unity's Animation Rigging package after MxM is already setup.

- **OnIdleTriggered** - This callback is called every time the character transitions into an idle state. Can be used to trigger custom gameplay logic whenever the player is idle. (Alternative use MxMAnimator.IsIdle to check if the character is currently idle)

- **OnLeftFootStepStart -** This callback is triggered every time a left footstep is triggered through the MxMAnimator. See section on 'Utility Tags' for more details on tagging footsteps. Note that any function specified for this callback must take a parameter of type 'FootStepData'.

- **OnRightFootStepStart -** This callback works the same as OnLeftFootStepStart but it is for the right footstep instead.

- **OnEventChangeState -** This is called every time a running event changes state. The state, the new state that has been entered will be passed as an enum EEventState:
    - EEventState.Windup
    - EEventState.Action
    - EEventState.Followthrough
    - EEventState.Recovery

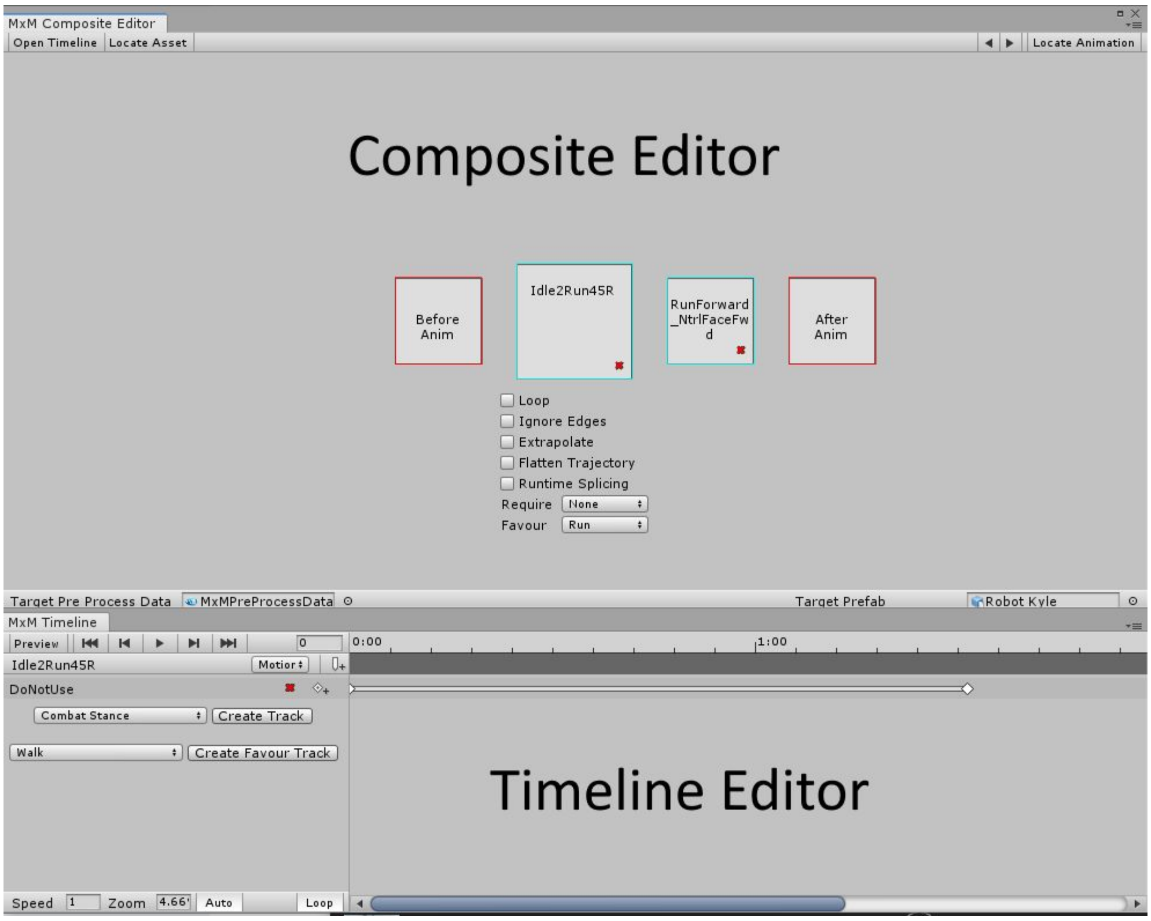
- **OnEventComplete -** Called every time an event is completed, either by reaching the end of the recovery State or being excited with motion during the recovery state. Events that are interrupted / cancelled by the user will not trigger this callback.

- **OnEventContactReached -** This is called every time a contact point has been reached during an event. The contact data will be passed with this callback.

3 Animation Composites

In this section we will go further in detail on Animation composites, how to configure them to suit your needs and how to add events and tags. Mastering animation composites is one of the keys to making cut clips work. It is also a vital step to making motion matching do more than just locomotion.

To edit an animation composite, simply double click on it in the animation composite list of the pre-processor. This will open two editors, the composite editor and the timeline editor.



Require: None
Favour: Run
Target Pre Process Data: MxMPreProcessData
Target Prefab: Robot Kyle
Idle2Run45R
Combat Stance
Walk
Speed: 1
Zoom: 4.66

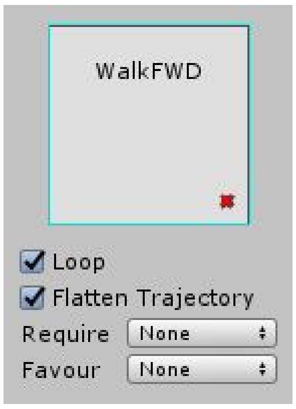
Figure 3.1 'Animation Composite Editor Layout'

3.1 Clip Continuity and Configuration

First let's talk about clip continuity. In motion matching, it is very important that each recorded pose, from the pre-process phase, has enough animation in front of it and behind it to record the trajectory that you specify in the pre-processor. However, this becomes difficult with cut clips as the animation ends immediately after the action has occurred with no way of determining the future.

There are a number of ways to overcome this shortcoming of cut clips in MxM and they are detailed as follows:

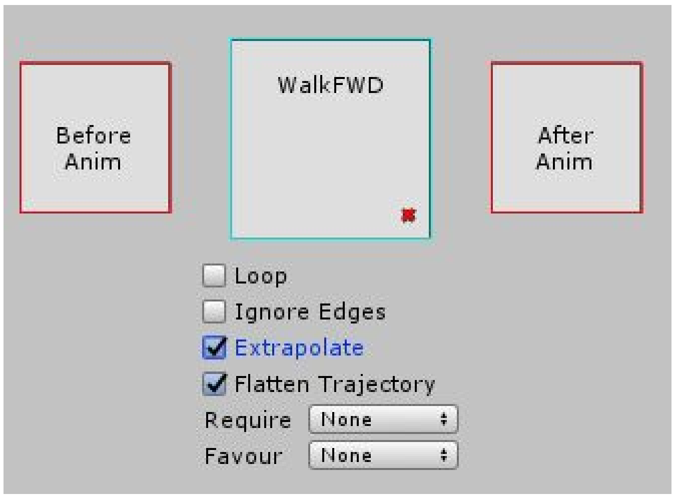
Looping Clips - If a particular cut clip is a looping clip, then the composite editor will automatically detect that and check the looping toggle as shown in Figure 3.2. In the case of looping clips, the past and future trajectory can be taken from the same clip as it loops.



☑ Loop
☑ Flatten Trajectory
Require None
Favour None

Figure 3.2 'Looping Animation Clip Composite'

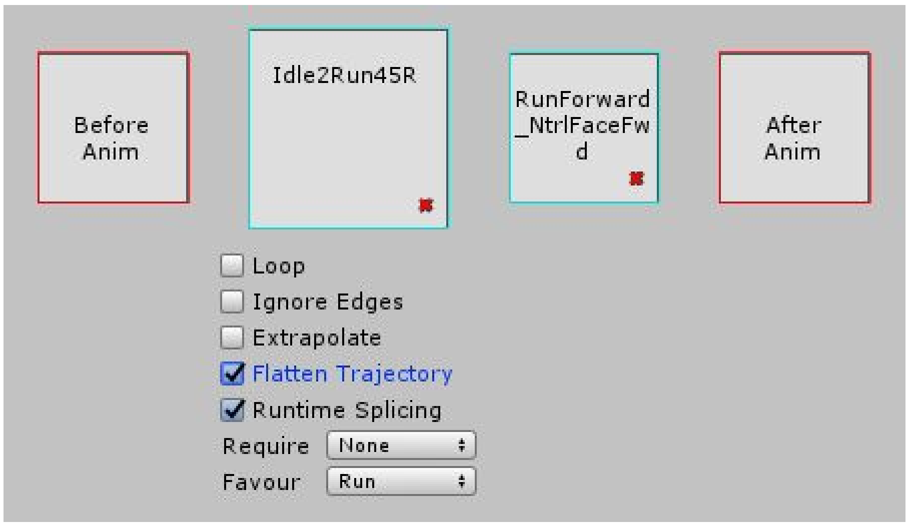
Extrapolating Trajectory - One easy option is to enable trajectory extrapolation as shown in Figure 3.3. If this is checked, the motion state at the end and beginning of the clip will be extrapolated to estimate a future trajectory. While this is not perfectly accurate, it is usually good enough.



☐ Loop
☐ Ignore Edges
☑ Extrapolate
☑ Flatten Trajectory
Require None
Favour None

Figure 3.3 'Extrapolate Trajectories'

Trajectory Borrowing - The best way to get a good trajectory is to borrow it from another clip. To specify a clip to borrow from, drag and drop it into the appropriate box to the left or right of the central box which contains our primary clip (Figure 3.4). The 'After Clip' box is for future trajectory and the 'Before Clip' box is for past trajectory. Ensure that neither 'loop', 'extrapoloate' or 'ignore edges' is checked.



- [ ] Loop
- [ ] Ignore Edges
- [ ] Extrapolate
- [x] Flatten Trajectory
- [x] Runtime Splicing

Require: None
Favour: Run

Figure 3.4 'Borrowing future trajectory from another clip'

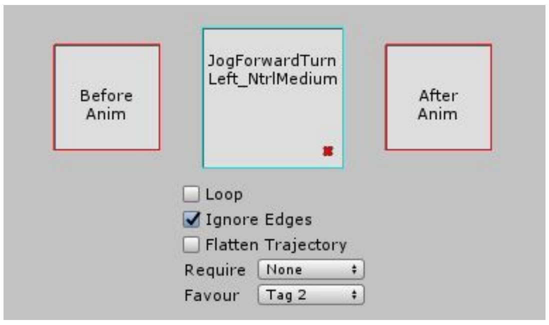
You will note that as you add a clip to either of these boxes, an additional empty box will appear. You can add as many animations as you like until you have enough to cover your trajectory needs.

Note: If an animation ends in an idle state there is no need to borrow from a different clip as you want the future trajectory to be static anyway. This same principle applies to the beginning of the clip as well. Save time by only borrowing when you need to.

Runtime Splicing - When an after clip is specified, as shown in Figure 3.4. The 'Runtime Splicing' option can be checked so that at runtime, the two clips will automatically be played together one after the other. This can be useful for move-start type animations as it reduces jitter caused by lacking continuity of cut clips.

Note: Do NOT use runtime splicing if the primary clip does not flow perfectly into the 'after clip'

Ignore Edges - This setting should only be used with long mocap takes which are continuous through a number of actions. If this toggle is checked, the system will mark all poses near the beginning and end of the clip with the 'DoNotUse' tag. Since their trajectory will not be created properly, they will be ignored.



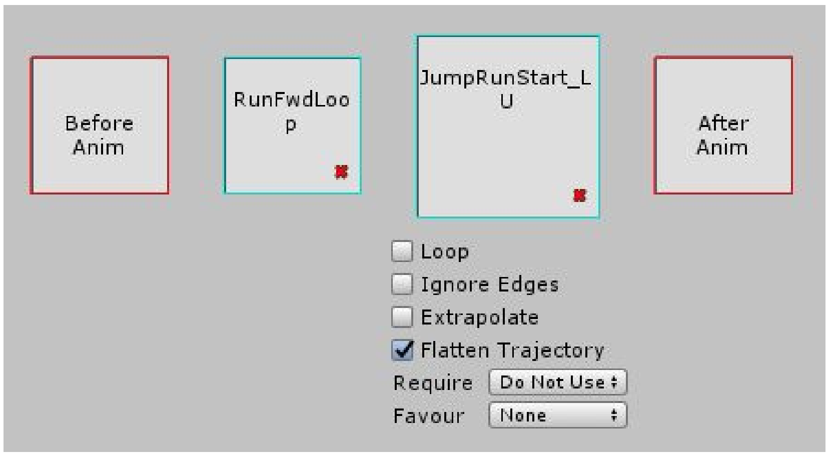
- [ ] Loop
- [x] Ignore Edges
- [ ] Flatten Trajectory

Require: None
Favour: Tag 2

Figure 3.5 'Ignore edges for mocap clips or long takes'

Note: Do not use this setting for cut clips.

Flatten Trajectory - By checking the ‘Flatten Trajectory option on a composite, it tells the pre-processor to generate a trajectory with no value on the ‘y’ axis. This can be useful when your runtime trajectory generator does not produce any variation in the ‘y’ and therefore, matching any other trajectory with a ‘y’ becomes pointless and mis-leads the motion matching algorithm. This is most useful for jump event animations.



- [ ] Loop
- [ ] Ignore Edges
- [ ] Extrapolate
- [x] Flatten Trajectory

Require: Do Not Use
Favour: None

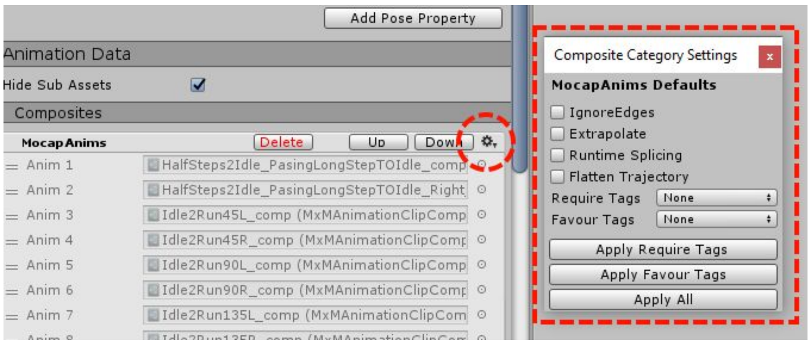
Figure 3.6 ‘Flatten Trajectory Option’

3.2 Global Tags

The last major setting in the clip composite editor is the global tags and global favour tags. These are the default tags that will be applied to all poses generated from this clip in their respective categories (i.e. require and favour). It is recommended to use this for things like stances where it is likely that the entire animation clip will be done in a single stance.

3.3 Default Composite Settings

It can become very tedious to open up every composite and change each setting individually. Therefore, there is a way to change the composite settings in bulk on a ‘per category’ basis. In the top right corner of each composite category there is a small cog icon. Clicking on this icon will open up a floating settings window as shown in Figure 3.7.



Hide Sub Assets: ✓

Anim 1	HalfSteps2Idle_PasingLongStepTOIdle_comp
Anim 2	HalfSteps2Idle_PasingLongStepTOIdle_Right
Anim 3	Idle2Run45L_comp (MxMAnimationClipComp
Anim 4	Idle2Run45R_comp (MxMAnimationClipComp
Anim 5	Idle2Run90L_comp (MxMAnimationClipComp
Anim 6	Idle2Run90R_comp (MxMAnimationClipComp
Anim 7	Idle2Run135L_comp (MxMAnimationClipCom

MocapAnims Defaults

- [ ] IgnoreEdges
- [ ] Extrapolate
- [ ] Runtime Splicing
- [ ] Flatten Trajectory

Require Tags: None
Favour Tags: None

Figure 3.7 ‘Composite Category Settings

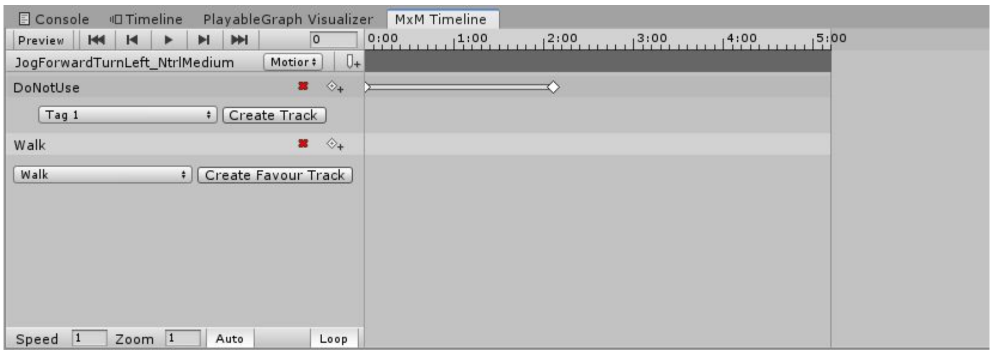
These settings mirror those in the composite editor. By changing these settings any new composite created within a category will inherit these default settings.

Any changes to these settings do not automatically apply to existing composites within the category. However, they can be applied by clicking on one of the buttons on the settings window as follows:

- *Apply Require Tags* - Will apply the default set 'require tag' to all composites in the category
- *Apply Favour Tags* - Will apply the default set 'favour tag' to all composites in the category
- *Apply All* - Will apply all default settings to all composites in the category

3.4 The Timeline Editor

The timeline editor, Figure 3.7, is linked to the composite editor (and blend space editor) and is used to preview and markup animations with events and tags on a granular level. You will notice that it resembles the standard Unity animation editor, and for the most part it operates in a similar way.



JogForwardTurnLeft_NtrlMedium

DoNotUse

Tag 1

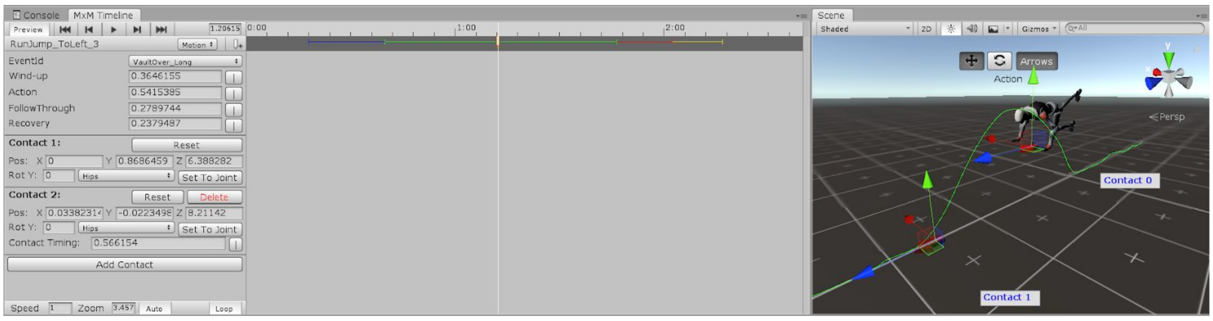
Walk

Walk

Speed 1 Zoom 1

Figure 3.8 'Timeline Editor'

To preview the animation, click on the 'Preview' button on the top left of the editor. If your scene is unsaved, you will be prompted to save it and then a preview scene will open showing your character animation and drawing it's trajectory on the ground. You can press 'Play' or scrub the timeline bar back and forth to view the animation.



1.20615

RunJump_ToLeft_3

EventId: VaultOver_Long
Wind-up: 0.3646155
Action: 0.5415385
FollowThrough: 0.2789744
Recovery: 0.2379487

Contact 1:
Pos: X 0 Y 0.8686459 Z 6.388282
Rot Y: 0 Hips

Contact 2:
Pos: X 0.0338231 Y -0.0223498 Z 8.21142
Rot Y: 0 Hips
Contact Timing: 0.566154

Speed 1 Zoom 3.457

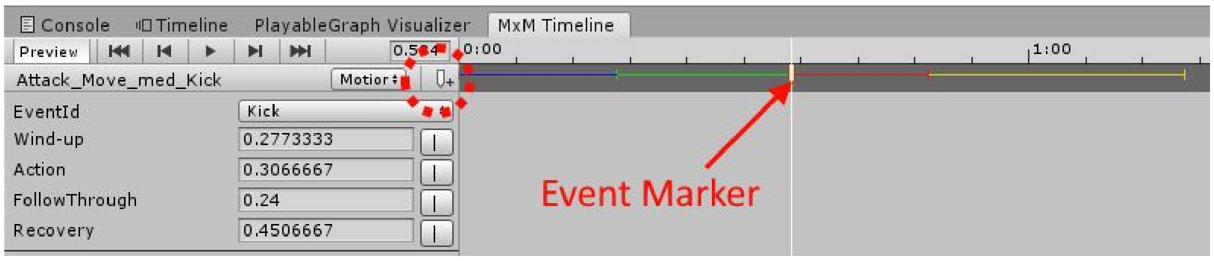
Figure 3.9 'Timeline Editor Preview'

3.4.1 Events

An event is some kind of action that you want the player to perform. Some examples of events include jumping, vaulting, attacking or pulling a lever. In all these cases we are playing a single animation once and then returning to locomotion.

To add an event, first scrub the timeline to the point in the animation where you want the event to occur. This can sometimes be arbitrary but often it is best to set the event to a point where contact is made with an object. For things like a sword hit, set the event to the point in time where you ideally want the sword to hit the enemy.

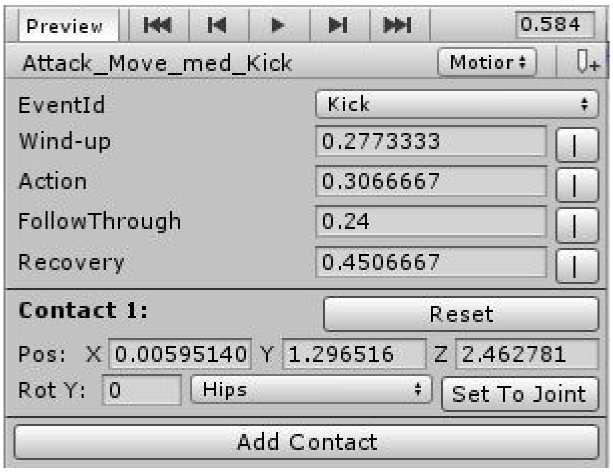
Click on the event button (Figure 3.9) and an event marker will be added on the event bar of the timeline. The event can be selected by clicking on it. It can also be dragged along the timeline to change it's time.



EventId	Kick
Wind-up	0.2773333
Action	0.3066667
FollowThrough	0.24
Recovery	0.4506667

Figure 3.10 'Adding an Event Marker

When selected you will notice the area on the left of the editor will change to reflect the data of the selected event (Figure 3.10). Modify the settings appropriately. Each setting description is explained below.



EventId	Kick
Wind-up	0.2773333
Action	0.3066667
FollowThrough	0.24
Recovery	0.4506667

Contact 1:

Pos: X 0.00595140 Y 1.296516 Z 2.462781

Rot Y: 0 Hips

Figure 3.11 'Event Data'

Event Id - This identifies the event and allows you to call an event during runtime based on your gameplay. These event IDs are created in your pre-processor as explained in Section 2.1.6.

Windup - Often actions will have a windup phase as the character winds up to anticipate the action. The value here specifies how long the windup should be. The windup phase can be visualised on the event marker by a blue whisker as shown in Figure 3.11. It is usually easier to set the windup point by setting the time scrubber to a specific point and clicking the | button.

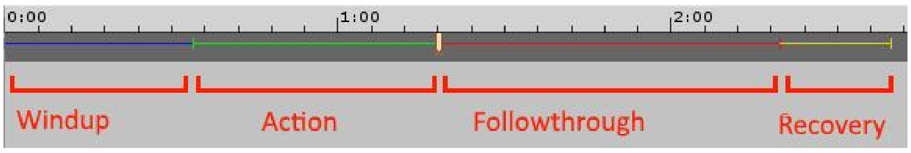


Figure 3.12 'Life of an Event'

The windup range is very important. it tells the system that it can start the event animation at any pose within the windup range. In short, the system will pick the best pose within the windup period and initiate the event from there. If you want the event to always initiate from the same point you can set the windup to 0.

Action - This is the part of the event where the character is committed to the action and is moving forward to perform the event. Essentially it is a compulsory part of the action before the event occurs. It is represented by a green whisker on the event marker.

Follow Through - Most actions have a follow through period that must occur to make the action valid or realistic. In the case of a sword swing this is the follow through of the sword swing after hitting. The follow through is represented by a red whisker on the event marker and it is also a compulsory part of the animation to play to make the animation valid.

Recovery - The recovery phase is the last phase of an action and it represents the animation required to return to the normal state of idle or whatever movement. The event can be automatically exited at any point during the recovery phase based on player input if desired. If there is no movement input during the recovery phase then the animation will play out, but if there is motion input, the locomotion will be matched in early and the recovery will be cancelled.

Event Contact - This is the position where the event takes place. If the event has a world contact point (e.g. touching a ledge while vaulting, or the sword hitting), the event position should be set to that point. The event position can easily be snapped to bones from the editor (see Figure 3.12) and even moved in the scene view via a gizmo. Make sure you are on the same frame as the event marker when setting the position so it is as accurate as possible.

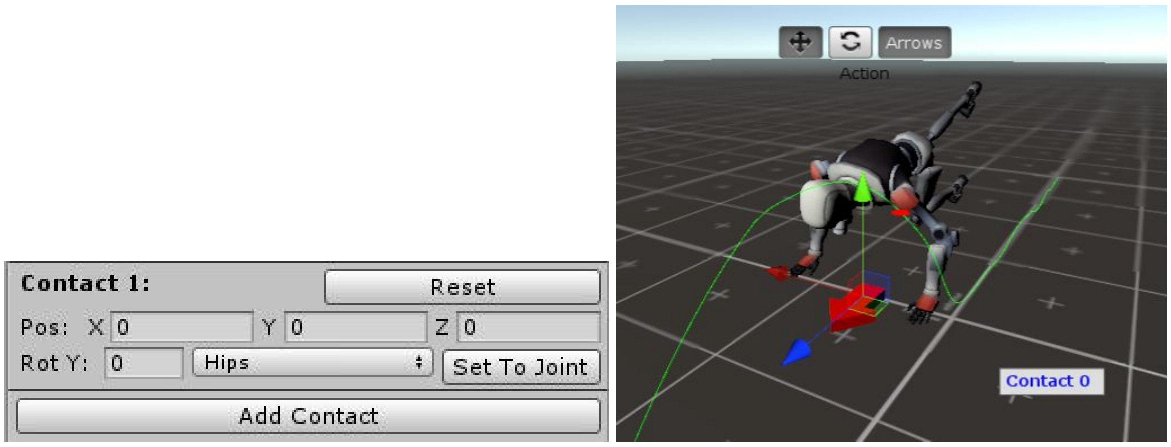

Contact 1:
Pos: X 0 Y 0 Z 0
Rot Y: 0 Hips

Figure 3.13 'Setting Event Position'

<u>Note:</u> The RotY field is for rotation warping. Rotation warping only ever occurs around the Y axis as it is extremely unusual to have a character 'root' to rotate around any other axis.

<u>Note:</u> It is standard practise to mark all poses within an event scope (whisker to whisker) with the DoNotUse tag as you don't want them playing out of context.

Multiple Contacts

It is possible to set up events with multiple contacts to enable complex parkour mechanics like wall running. Even a simple vault action might have multiple contacts. For example, 1 contact when the hand touches the vaultable object and another contact when then feet hit the ground.

To add additional contacts, click the 'Add Contact' button as shown in Figure 3.11. You can add as many as is needed and each will have its own gizmo in the scene view. Please note that contact points should be sequential and not occur at the same time. If you have multiple hand contacts at the same time, for example, you can either place the contact point between them or just pick one and manually use IK for other simultaneous contacts.

Figure 3.13 shows the timeline editor when there are multiple contacts. For the whiskers each contact is considered a continuation of the 'Action' phase of an event.

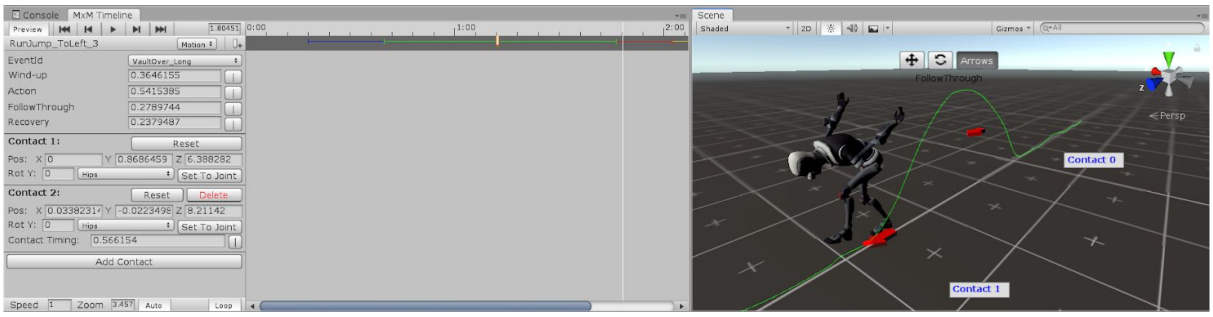

RunJump_ToLeft_3 Motion
EventId VaultOver_Long
Wind-up 0.3646155
Action 0.5415385
FollowThrough 0.2789744
Recovery 0.2379487
Contact 1:
Pos: X 0 Y 0.8686459 Z 6.388282
Rot Y: 0 Hips
Contact 2:
Pos: X 0.0338231 Y -0.0223498 Z 8.21142
Rot Y: 0 Hips
Contact Timing: 0.566154
Speed 1 Zoom 3.457

Figure 3.14 'Event data area with multiple contacts'

To handle multi contact events at runtime, please see section 8.5.

3.4.2 Tags

There are four different types of tags that can be used with MxM including:

- 'Require Tags'
- 'Favour Tags'
- 'Utility Tags'
- 'User Tags'.

Each type has their own specific purpose and are explained in section 3.3.3 through 3.3.6. General functionality of adding and removing tags / tracks are as follows:

Switching Between Tagging modes

You can switch between timeline modes from the drop down menu on the second toolbar on the left panel. Three of the modes are related to tagging and they are as follows:

- *Motion Matching* - Add 'Require Tags' and 'Favour Tags'
- *Utility* - Add 'Utility Tags'
- *User* - Add 'User Tags'

Adding and Removing Tag Tracks

Tags are added via tracks. Each track is linked to a chosen tag. To add a track, choose the tag you want on the left panel of the timeline editor and click the 'Create Track' button. You can only have one track for each tag.

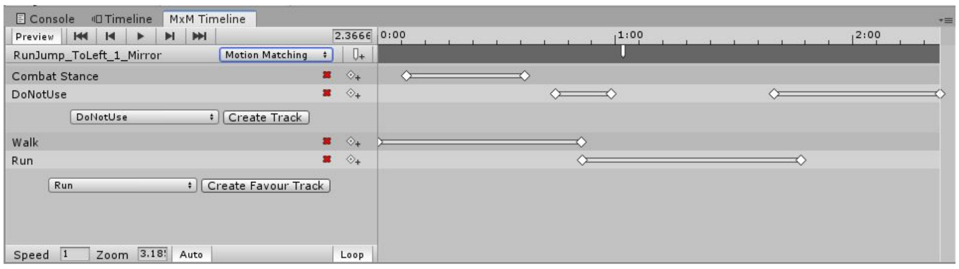


Figure 3.15 'MxM Timeline Tagging Interface'

To delete a track, simply click the cross button on the track.

<u>Note:</u> You cannot create or delete utility tracks. These tracks are fixed. See Section 3.3.6 for more details

Adding Tags

To add a tag to a track, click on the add tag button (next to the red cross) on the track that you desire to add it to. You can add multiple tags to a track at different locations.

Selecting and Moving Tags

Tags can be selected by clicking on them. You can drag the entire tag along the track or just manipulate the end handles to change the scope of the tag. Any poses that fall within the tagged range will be marked with the appropriate tag during pre-processing.

3.4.3 Require Tags

Unlike events, there are only a limited number of ‘require’ tags that can be used. There are 32 tags in total but 2 are reserved for MxM. The 30 remaining tags can be named in the ‘Tags and Events’ - ‘Tags’ foldout of the pre-processor inspector, and used for whatever you like.

For the most part, required tags can be used to specify styles and stances. For example, you may have a combat style stance, in which case you will want to tag all animations with a combat style with this tag. If an entire animation clip should have the tag, use the global tag in the composite editor instead.

Another example of a tag is ‘Crouch’. The system needs to be able to differentiate between animations of different kinds and tagging is how this is achieved. Think of it as a bool variable in a mecanim state machine. When the player presses the crouch key, your gameplay needs to tell MxM that the ‘Crouch’ tag is required. See section 8.9 how to do this through code.

The DoNotUse Tag
The pre-process system will automatically mark some poses as DoNotUse in varying circumstances. However, users can manually add DoNotUse tags to their animations to iteratively improve the output.

If there is a bad section of animation, it can be marked as ‘DoNotUse’. Alternatively, if you notice some animations frames are being used that you don’t like, simply mark them as DoNotUse and the system will be forced to use something else.

The DoNotUse tag can be a powerful iterative tool for polishing and refining your motion matching.

3.4.4 Favour Tags

Favour tags are similar to require tags. However, they work differently at runtime. If a user specifies that a favour tag is required, only poses with that tag will be played. However, if the user specifies that a tag should be favoured (with a favour factor provided), poses that have the matching favour tags will have their cost reduced, meaning they will be more likely to be chosen.

The amount of favour applied to poses with favour tagging is dependent on the ‘Favour Tag Mode’ set in the MxMAnimator Component. The three modes include:

- Exclusive - Poses must have all the requested favour tags to be given favour
- Inclusive - Poses must have at least on requested favour tag to be given favour

- Stacking - Poses will be given exponentially more favour based on the number of requested tags that it matches.

As favour tags and require tags are mutually exclusive, there is a separate set of 32 favour tags that can be defined in the pre processor inspector and tagged in the timeline window.

An example use case for favour tags is to help differentiate between movement speeds. While the system typically has no trouble differentiating walking and running automatically, the more movement speeds there are in between can cause some 'confusion' resulting in animation popping between different speeds. Using favour tags, different speeds can be tagged and gameplay logic can be used to specify which speed is favoured, essentially helping the system to make the right decision.

If used well, favour tags can provide better results than required tags and even replace them completely.

3.4.5 User Tags

Unlike require and favour tags, user tags have no automatic functionality. Rather, they are defined by the user and their functionality is also defined by the user. At runtime, a user can query the MxMAnimator for all user tags active on the current animation pose. The user can then run custom logic based on what tags are active. For example, this could be used to enable, and disable gravity or even collisions during specific actions.

To access user tags, choose 'User' from the dropdown menu on the toolbar of the left panel as shown in Figure 3.14.

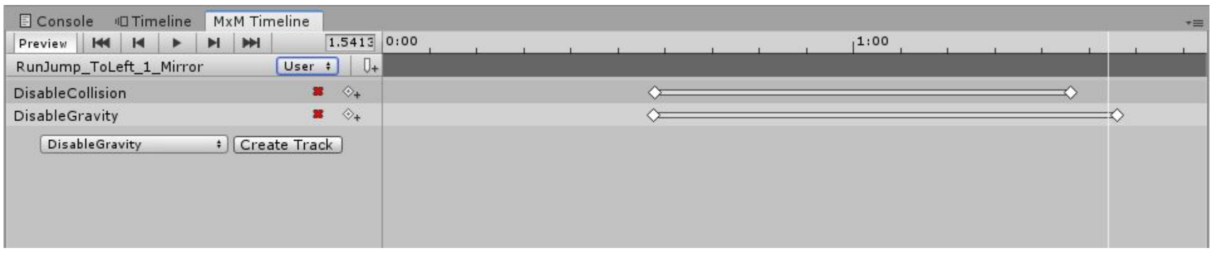
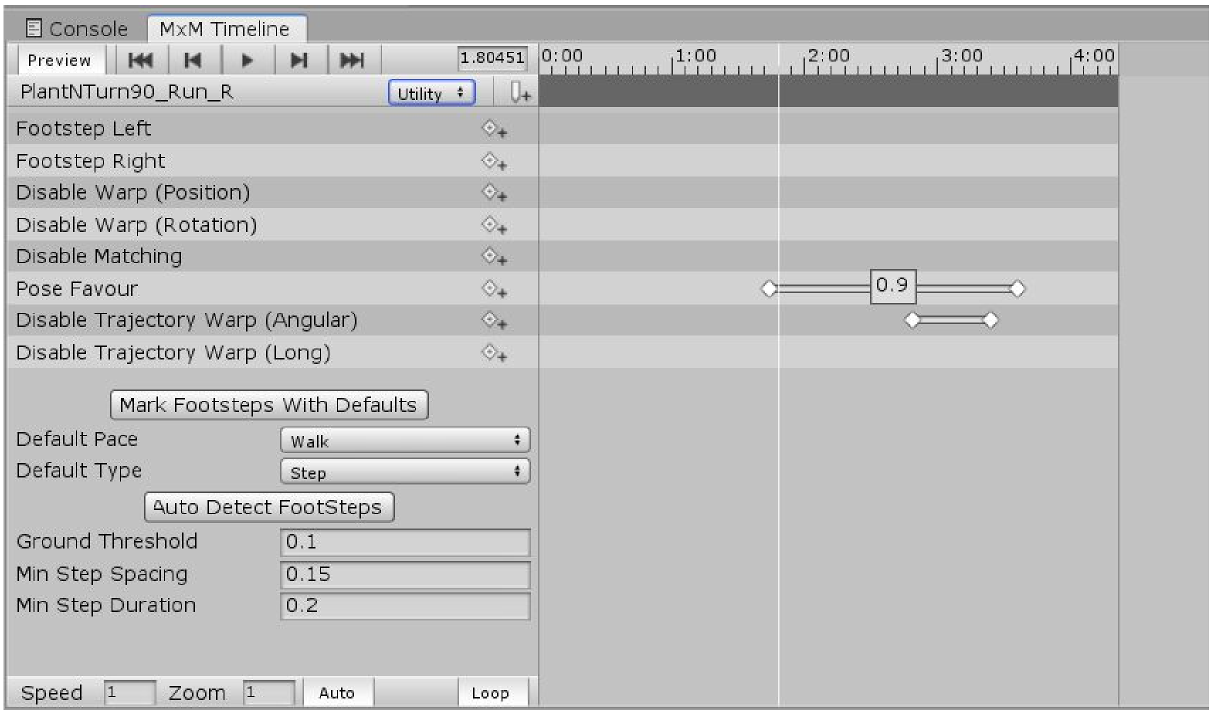


Figure 3.16 'Timeline User Tags'

3.4.6 Utility Tags

The third mode of the timeline editor is for utility tags.



Default Pace: Walk
Default Type: Step
Ground Threshold: 0.1
Min Step Spacing: 0.15
Min Step Duration: 0.2
Speed: 1
Zoom: 1

Figure 3.17 ‘Utility Tagging’

There are 7 Utility tag tracks including:

- Footstep Left
- Footstep Right
- Disable Warp (Position)
- Disable Warp (Rotation)
- Pose Favour
- Disable Trajectory Warp (Angular)
- Disable Trajectory Warp (Longitudinal)

Footstep Left - This tag track can be used to tag up left footsteps. Each tag represents a step, with the beginning of the tag being the start of the step and the end of the tag being the end of the step. Footsteps can also have metadata attached to them. The default meta data is shown in Figure 3.17 but each tag can be selected to specify individual tag metadata.

Footstep Right - This track works the same as the left footstep track but for the right foot.

Automatic Footstep detection - The utility timeline comes with an automatic footstep detector to help speed up the process of tagging footsteps. This saves a lot of time when detecting footsteps. However, some manual cleanup may be required after footstep detection.

To auto detect footsteps, simply set the appropriate settings and click the ‘Auto Detect FootSteps’ button. The Footstep Left and Footstep Right tracks will be populated with tags that can be edited.

Note that by default all detected footstep tags will be given metaData as set with the ‘Default Pace’ and ‘Default Type’. The meta data can be changed on mass for both tracks by choosing ‘Mark Footsteps With Defaults’ or individually by selecting the tags and modifying the meta data.

Warning: Auto detecting footsteps will clear any previous footstep work.

Auto footstep detection works basically by checking if the toe is moving at a speed below a threshold (0.1m/s). Additional settings can help auto-detection as described below::

- Ground Threshold - Feet must be under this height to be able to detect a footstep (Try 0.25)
- Min Step Spacing - the minimum time spacing between footsteps (detected footstep that are too close will be combined)
- Min Step Duration - the minimum duration of a footstep for it to be considered valid (footsteps that fail to meet this minimum duration will be deleted)

Adjust the settings until most footsteps are detected properly and then use those same settings for your other animations.

Disabling Warping: Animation warping is the technique used to ensure that world contacts (e.g. hand placement during a vault) are met regardless of how well your animation matches the environment. It does this by warping the root motion. During certain frames, warping is undesirable, the disable warping tags, both positional and rotational, can be used to specify which frames should not allow warping.

An example of where you might want to disable warping is during a vault. Warping should only occur while the character is in the air and not when the feet are on the ground or when the hand is planted. By marking these ‘planted’ sections to disallow warping, foot or hand sliding can be avoided for a much more realistic motion.

For rotational warping, it is sometimes ok to warp while only one joint is planted. However, there are no hard and fast rules and the choice is up to you.

Pose Favour - This track is very powerful and it allows favouring of specific animation poses. Unlike the ‘DoNotUse’ tag which causes the system to completely ignore poses, pose favour tags allow poses to be tested but with additional or reduced weighting depending on the tag settings.

When poses are assessed they are given a cost. The higher the cost the less likely it is to be chosen as the next animation pose. The calculated cost is multiplied by the pose favour to determine the final cost. Therefore, higher numbers mean the pose is less likely to be chosen while lower numbers increase the chance of the pose being chosen.

Use pose favouring to finesse and curate your animation database. Perhaps there are some poses or animations you want to occur sometimes but not as often as you are currently seeing them. This is a good usage of pose favouring.

Note: It is recommended to always apply favour to your movement loops (e.g. RunFwdLoop) between 0.8 - 0.9 to ensure smooth looping playback.

Disable Trajectory Warp (Angular) - This utility track disables trajectory warping for tagged poses in animations. This is useful for animations like plants or sharp turns where angular error warping may reduce animation fidelity.

Disable Trajectory Warp (Long) - This utility track disables trajectory warping for tagged poses in animations. Use as required.

4 Animation Idle System

Motion matching typically does not handle idle animations very well because they usually contain very little motion. This is ok because during idles we don’t really want to change clips constantly and it is better to stick with an idle until it is complete, or until movement is detected.

MxM comes with a dedicated idle system, built into the MxMAnimator. It ensures that poses are matched in and out of the idle but simply plays the idles inbetween without matching.

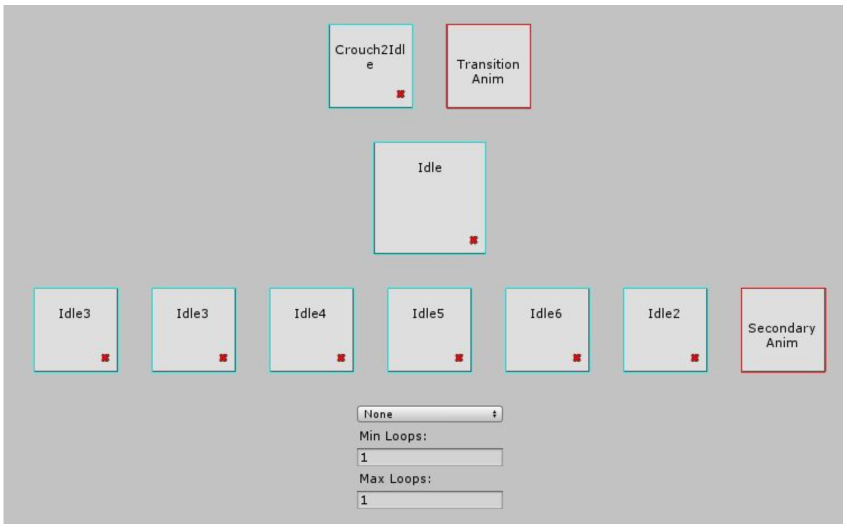
4.1 Idle Sets

An idle set is a set of idle animations that go well together. As a minimum, each idle set has a ‘Primary Idle’ animation. A primary idle is your normal idle animation without any flair and it will be played most of the time.

Idle sets can also have several ‘Secondary Anims’ and several ‘Transition Animations’
A secondary idle is a special idle where the character may kick the ground, stretch their arms or do some other benign action while idle.

Transition Animations allow you to specify animations that transition from other stances to this idle (e.g. IdleToCombat). The motion matching backend will automatically pick the best transition idle, provided the character isn’t already in that idle state. It can be beneficial to add move stop animations to your transition animations just to make sure that the switch to idle doesn’t cut stop animations short.

If you created the idle set by dragging an idle animation into the ‘Idle Sets’ foldout of the pre-processor, then the ‘Primary Clip’ will automatically be set and it’s name shown in the central box. Secondary idles can be dragged into the smaller boxes beneath the primary, while transition clips can be dragged into the smaller boxes above the primary (Figure 4.1).



None
Min Loops:
1
Max Loops:
1

Figure 4.1 ‘Idle Set Editor’

When an idle is detected, a starting pose will be chosen within your primary idle or one of the transitions clips. The motion matching backend will automatically pick the best point within those animations to start. Once a transition is complete (if chosen) the primary idle clip will be played for a random number of loops between the min and max loop values specified in the ‘Idle Set Editor’ (Figure 4.1).

Once this semi-random number of loops have passed, one of the secondary idle animations will be chosen and played. The selection of secondary idles is random but the system does ensure that the same secondary idle will not be played twice in a row if there is more than one.

4.2 Global Tags

Much like animation clip composites, idle sets can be given global tags. This is set by the dropdown button shown in Figure 4.1.

Each idle set should have a different global tag as you would have different idles for different stances. When the system detects that the character is idle, the idle set used will be the one that has the required tags currently set in the MxMAnimator.

It is also possible to have multiple idle sets for the same stance. This is useful if your character has a staggered idle stance (i.e. one foot slightly forward, one foot slightly back). The motion matching can automatically pick the closest one for you.

Note: Staggered stances are highly recommended as they increase responsiveness as the player never has to take an additional step to stop in the correct idle stance. This is a technique used in many AAA games.

5 Blendspaces

Traditionally the biggest weakness of motion matching is that it can only play animations that it is given. Aside from rapid fading between animations, there is little in the way of parametric blending. For quality mocap data this is not a problem but for cut clips with little coverage it has some significant adverse side effects where animations seem to jump and pop, particularly banking or turning animations.

Motion Matching for Unity is the first motion matching system to address this coverage issue by implementing blendspaces into the motion matching algorithm.

5.1 Creating and Editing a Blendspace

To create a blendspace, navigate to the 'Animation Data -> Blend Spaces' tab of your MxM Pre-Processor and drag in an animation to create a blendspace sub-asset. Double click on the new blendspace in the list to open the blendspace editor (Figure 5.1).

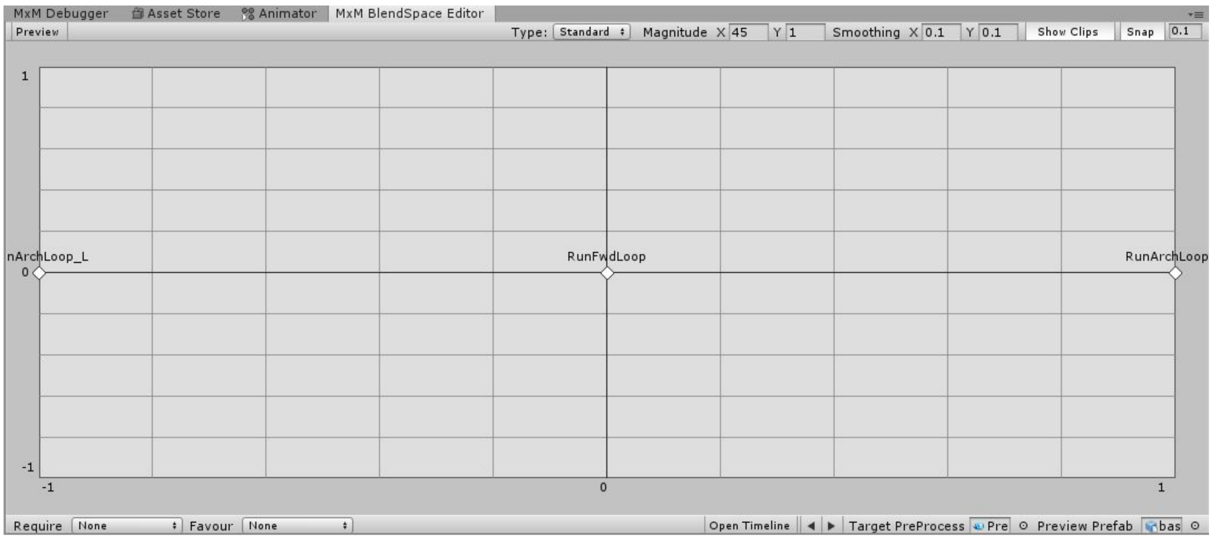

Type: Standard
Magnitude X 45 Y 1
Smoothing X 0.1 Y 0.1
Snap 0.1
Require None
Favour None
Target PreProcess Pre
Preview Prefab bas

Figure 5.1 The blendspace editor

Blendspaces work very much like blend trees in mecanim except they are set out in a normalized grid. Drag and drop animations into the grid to add them to the blendspace. You can click the preview button to view the results, and click / drag anywhere in the blendspace to move the position within the blendspace.

5.2 Blendspaces for Locomotion

Blend spaces can be used to significantly improve locomotion when cut clips are being used. However, part of the benefit of motion matching is also reducing the amount of blending that is required. In that way, blend spaces should be used in a way that is not destructive to the animation and preserves its quality. Therefore, the best way to set up blendspaces for motion matching is with bank sets instead of strafe sets as shown in Figure 5.2.

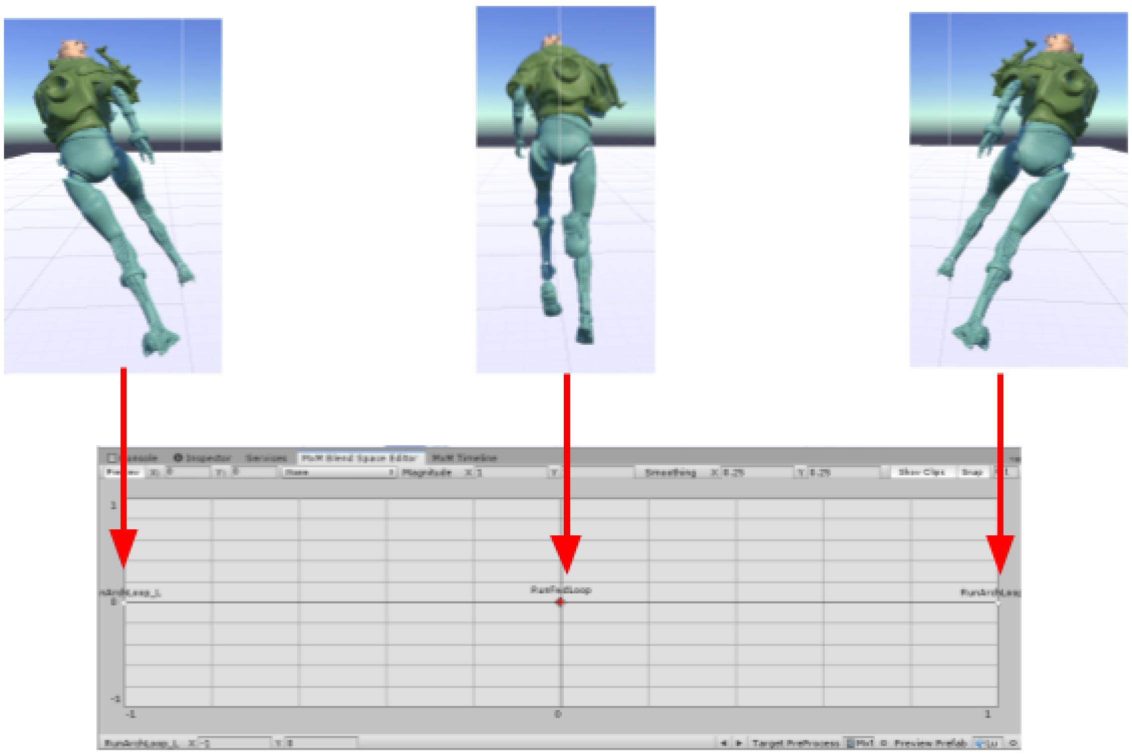


Figure 5.2 Blendspace ‘Banking Set’

For each forward moving animation that has an arcing counterpart, it is recommended to create a blendspace instead of a composite. The forward moving blendspace should contain the forward straight moving animation in the center (0, 0), with a left arc to the far left of the blendspace (-1, 0) and a right arc to the far right (1, 0) as shown in Figure 5.2.

By doing this, coverage is significantly improved compared to simply using composites as shown in Figure 5.3.

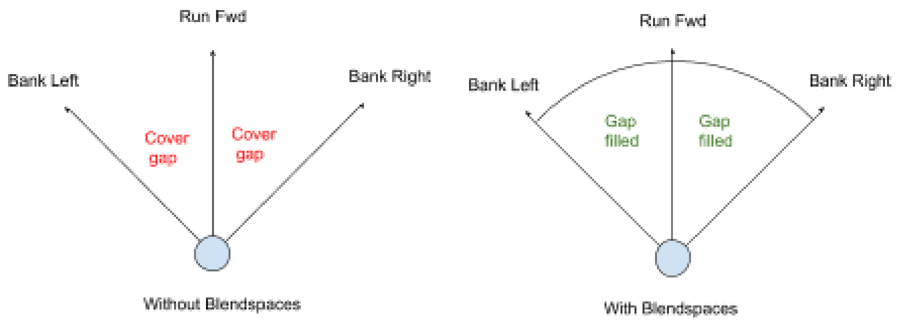


Figure 5.3 Blendspace coverage example

Note: It is important that all of the animations in a blendspace have the same play length and be in sync. Blendspaces are a feature for use with clean cut clips, not mocap data.

Strafing with Blend Spaces
A traditional strafe set in mecanim is done by adding all 8 directions of strafing into a single tree. However, as mentioned before, this can be very destructive to the animations causing poor quality, specifically leg crossover. This generally occurs when there are just too many animations with vastly different movement types being blended all at once.

To combat this, strafe sets should be split up into 4 separate blend spaces representing NSEW directions. Each blend space will have only 3 directions and should be setup as follows:

- North *(Scatter X = 0.05 to 0.1)*
    - MoveForward (0, 0)
    - StrafeLeft45 (-1, 0)
    - StrafeRight45 (1, 0)
- South *(ScatterX = 0.05)*
    - MoveBackward (0, 0)
    - StrafeLeft135 (-1, 0)
    - StrafeRight135 (1, 0)
- East *(Scatter Y = 0.05)*
    - StrafeLeft (0, 0)
    - StrafeLeft45 (0, 1)
    - StrafeLeft135 (0, -1)
- West *(ScatterY = 0.05)*
    - StrafeRight (0, 0)
    - StrafeRight45 (0, 1)
    - StrafeRight135 (0, -1)

Via this method, there will only ever be two animations blended at once but you will still be able to achieve every angle of strafing movement.

Note: This strafing method requires scatter mode on the Blend Spaces on particular axis. See Section 5.2.1 for more details.

5.2.1 Explicit Methodology (Recommended)

The recommended way to use blendspaces for locomotion with MxM is to convert it into a scatter space. With a scatter space, the blendspace will be sampled, assessed and pre-processed at several intervals within the blend space. This will generate many different angles of moving in between the limits of the blend spaces, simulating some of the coverage achieved by mocap. This allows for perfect matching within blend spaces and is the most accurate method while being the least prone to error.

To use a blendspace as a scatter space simply change its type to on of the scatter types:

- Scatter - The blend space will be sampled at specified intervals along the x and y axis forming a grid of samples. Only use this type if you have animations on the X and Y axis of the blend space.
- Scatter X - the blend space will be sampled at specified intervals along the X axis of the blend space
- Scatter Y - the blend space will be sampled at specified intervals along the Y axis of the blend space.

When a scatter type is chosen, the spacing interval can be selected in the subsequent settings as shown in Figure 5.4. The spacing is within the normalized blend space limits from -1 to 1. So a spacing of 0.05 will result in 40 samples along a single axis.

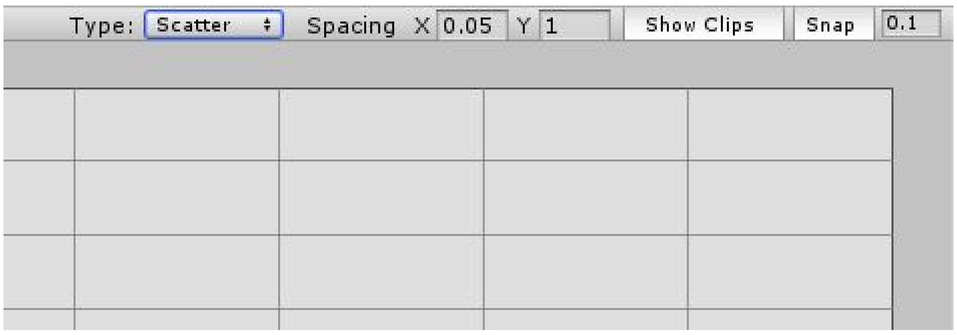

Type: Scatter
Spacing X 0.05 Y 1

Figure 5.4 ‘Scatter spacing settings on a blend space’

Note: It may be tempting to limit the spacing for performance reasons, however, as cut clip animations produce such fewer poses than mocap, it is recommended to be generous with the spacing interval in exchange for smoothness in this case.

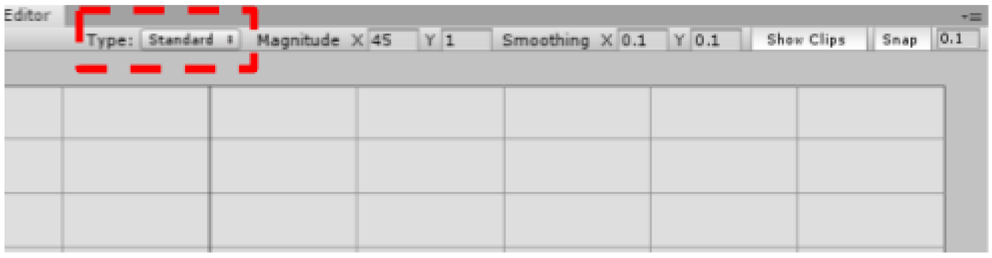
What to do with the Y axis

One potential use of the Y-axis for locomotion is to implement sloped movement. This is because, just like banking arcs, it is less destructive to the base animation. Note that your trajectory generator will need to take slopes into consideration for this to work.

Varying move speeds could also be used on the Y axis. However, it is not usually recommended as the varying speeds would need to be in sync to work and it may reduce the quality of the animation somewhat.

5.3 Triggered Looping Blendspaces

In addition to using blendspaces for locomotion coverage, blendspaces can be triggered to continuously play in a loop. This is good for actions that don't make sense to 'motion match' like gliding or falling for example. For blend spaces to be used like this, they need to be setup as 'standard' blend spaces as shown in figure 5.5



Type: Standard
Magnitude X 45 Y 1
Smoothing X 0.1 Y 0.1
Snap 0.1

Figure 5.5 'Standard blend space type'

A blend loop can be triggered through a function call to the MxMAnimator

```
void BeginLoopBlend(int a_blendSpaceId);
```

While a blendspace is playing, the position within the blendspace can be modified by calling the following functions:

```
void SetBlendSpacePosition(float a_x, float a_y, bool a_autoNormalize);
```

```
void SetBlendSpacePositionY(float a_y, bool a_autoNormalize);
```

```
void SetBlendSpacePositionX(float a_x, bool a_autoNormalize);
```

If you set the *'a_autoNormalize'* parameter to 'true' then you can pass in non normalized values which will be normalized automatically to fit in the blendspace based on the blendspace magnitude (explained in section 5.2.2).

Note: The Y axis can be left blank without any issues.

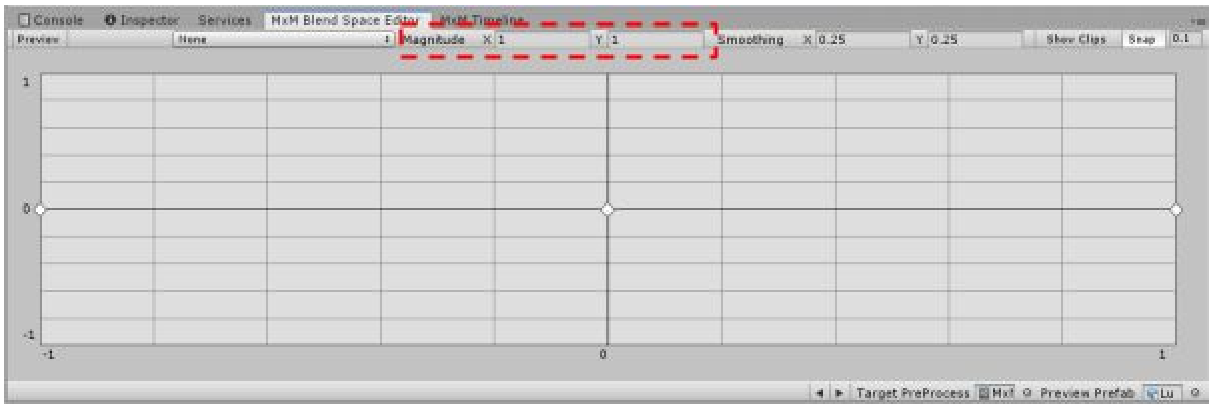
To end a blendspace loop and return to regular motion matching call;

```
void EndLoopBlend();
```

Note that while matching is not occuring during the blendspace loop, animations are matched coming in and going out of a blendspace so that there is the best possible transition.

Magnitude

Blendspaces operate in a normalized space from -1 to 1. However, the magnitude of the blendspace can be set in the top bar (Figure 5.6) of the blendspace editor. When using the blendspace setting functions, you can pass in the raw data, and the normalized position will be calculated based on the magnitude of the blendspace.



Magnitude X 1 Y 1
Smoothing X 0.25 Y 0.25
Snap 0.1

Figure 5.6 Blendspace magnitude

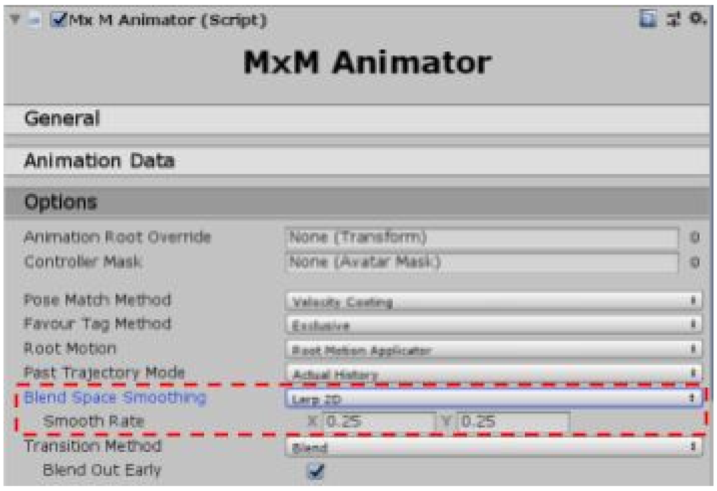
As an example, if you were to pass in the current slope the character is on, you might have a max slope of 30 degrees. In this case you would set the 'Y' magnitude of the blendspace as 30. Now when you call SetBlendSpacePositionY(currentSlope), the value of Y on the blendspace will be normalized based on the magnitude and the current slope you passed.

Smoothing

Jumping to positions instantly in a blendspace can cause jitter and undesired snapping. Therefore, blendspaces have a smoothing feature. The type and rate of smoothing can be set in the MxMAnimator inspector as shown in Figure 5.7 or by manually setting the following properties in the MxMAnimator component.

```
EBlendSpaceSmoothType DefaultBlendSpaceSmoothType
```

```
Vector2 DefaultBlendSpaceSmoothRate
```


Mx M Animator (Script)

MxM Animator

General

Animation Data

Options

Animation Root Override	None (Transform)
Controller Mask	None (Avatar Mask)
Pose Match Method	Velocity Costing
Favour Tag Method	Exclusive
Root Motion	Root Motion Applicator
Past Trajectory Mode	Actual History
Blend Space Smoothing	Lerp 2D
Smooth Rate	X 0.25 Y 0.25
Transition Method	Blend
Blend Out Early	✓

Figure 5.7 Blendspace smoothing settings

Smoothing Types:

- None - No smoothing
- Lerp - Both X & Y axis will be smoothed with the same smooth rates
- Lerp2D - X & Y axis will be smoothed with independent smooth rates
- Unique - X & Y axis will be smoothed with the same smooth rate dependent on the unique smooth rate specified for each blendspace.
- Unique2D - X & Y axis will be smoothed with independent smooth rates based on the unique smooth rate specified for each blendspace.

Unique blend rates can be set for each blendspace individually in the blendspace editor on the top bar (Figure 5.8).

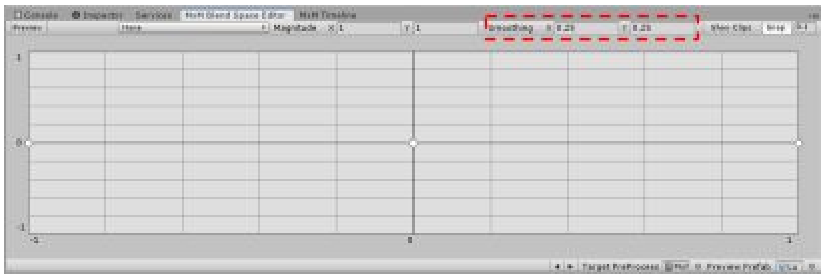


Figure 5.8 Blendspace unique smooth rates

Depending on the blendspace smoothing settings in the MxMAnimator and on the individual blendspaces, the position will smoothly transition to the desired position or snap instantly.

5.4 Tagging and Speed Manipulation

Tagging and speed manipulation for blend spaces works exactly the same as with animation composites.

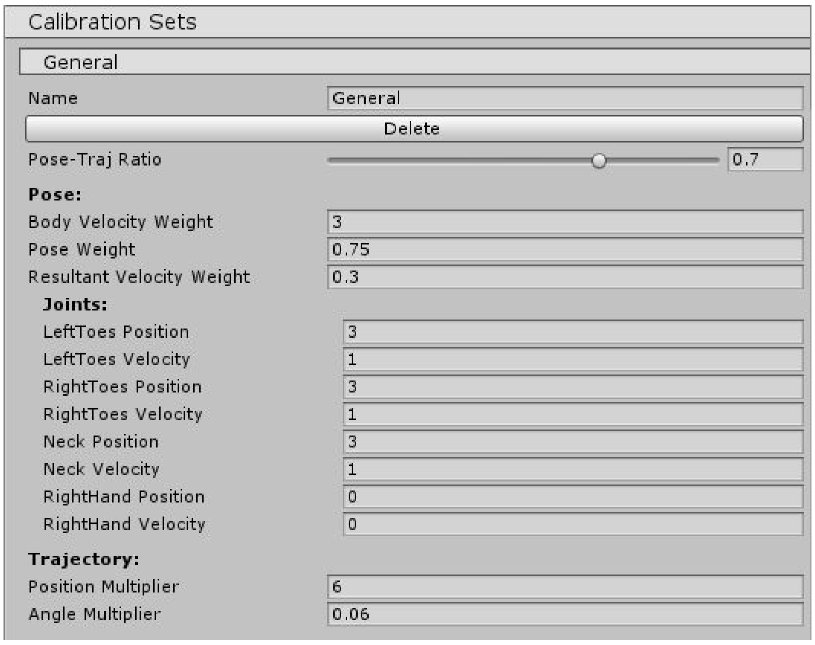
5.4 Events

Blendspaces currently don’t support events. The reason for this is that blendspaces demand that animations to be looping. Such animations are generally not suitable for events and therefore, it has not been included.

6 Calibration

In its simplest form, Motion Matching is a big cost function which compares the desired state of the character to all the animation poses generated in the pre-process phase. The result of the cost function is that the lowest cost pose gets chosen and blended in. However, how does it differentiate differences between positions, angles or velocities where their values can vary wildly. It's not exactly logical to compare a 1 degree angle difference to a 1 meter position difference.

This is where calibration comes in. We feed the system a set of multipliers which act as a weighting for different aspects of the pose and trajectory. In the case of our facing angles, the multiplier is usually very small (around 0.05) so that we can effectively normalise the cost of each aspect of the pose and trajectory.



Calibration Sets

General

Name	General
Delete	
Pose-Traj Ratio	0.7
Pose:	
Body Velocity Weight	3
Pose Weight	0.75
Resultant Velocity Weight	0.3
Joints:	
LeftToes Position	3
LeftToes Velocity	1
RightToes Position	3
RightToes Velocity	1
Neck Position	3
Neck Velocity	1
RightHand Position	0
RightHand Velocity	0
Trajectory:	
Position Multiplier	6
Angle Multiplier	0.06

Figure 6.1 'A typical calibration'

Calibration is performed in the calibrations foldout of the *'MxMAnimData'* asset inspector. Let's take a look at all the things that are matched for the pose and trajectory and discuss the logic being calibration multipliers.

6.1 Pose Calibration

The pose is made up of four major components including:

- Relative velocity of the character body
- Relative position of each matched joints to the root joint
- Relative velocity of each matched joints to the root joint
- Resultant velocity (from current pose to candidate pose)

The purpose of matching the character body velocity is that you don't want to pick a pose where the velocity is wildly different or your motion will not be fluid. You also have to consider that there is only one metric for character body velocity, so it's common to set this value quite high. If you set it too high though, it will start to affect responsiveness. You still want to be able to change body velocity to some degree.

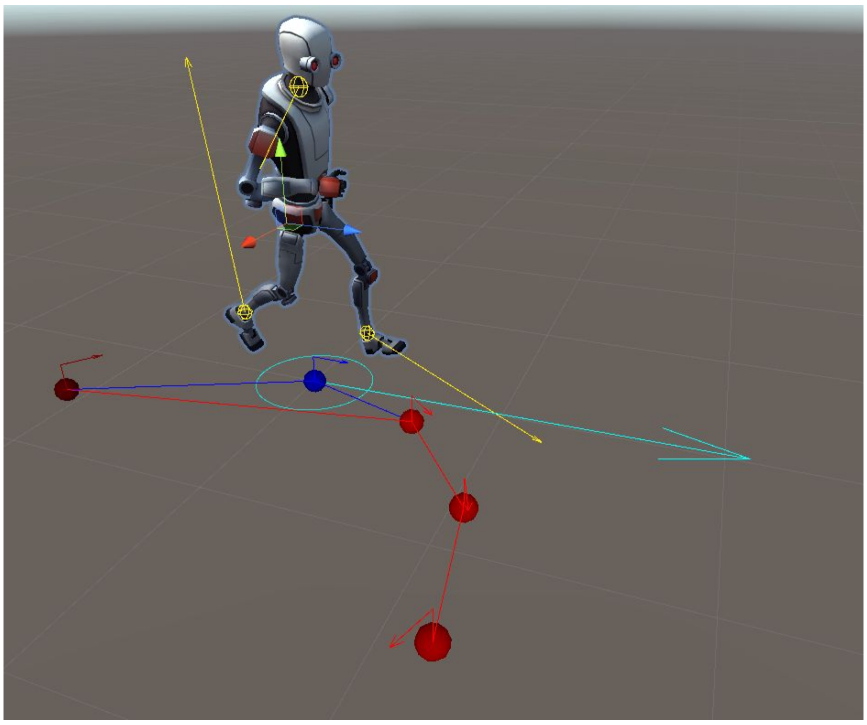


Figure 6.2 'Pose body velocity shown in blue'

The reason for matching joint positions and velocities is relatively obvious as we don't want the pose to change widely within a single frame. It's worth noting that joint velocity variations can be quite high compared to variations in joint position so it is recommended to weight positions higher. Note in Figure 6.3 how large the velocity vectors of joints can be.

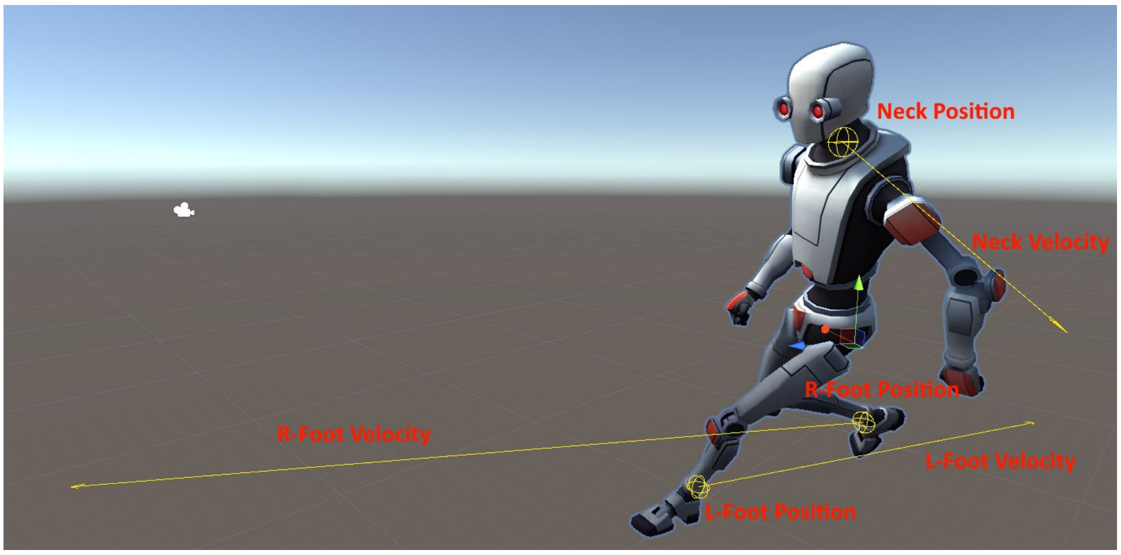


Figure 6.3 'Visualisation of pose data'

In some cases it may be helpful to weight matched joints that are not mirrored (e.g. the hips or neck) a little higher to compensate. If you are matching both feet you have costs associated for two joints that perform similar motion that is relatively in sync. Without an increase in weighting to the stabilizer joint it could have very little effect.

Resultant Velocity Costing
It's one thing to compare velocity vectors between joints. However, this doesn't prevent animation going backward. Two poses may have very similar velocities, but going from one pose to another might actually cause a 'resultant velocity' in a completely opposite direction. Therefore, the system also compares the cost (or difference) between the current pose joint velocity and the resultant velocity if it were to jump to a candidate pose.

Resultant velocity costing helps remove jitter and improve motion fluidity very significantly. It also prevents animations from running backward. However, it is extremely sensitive. A weighting value between 0.1 and 0.2 is recommended. If the character feels unresponsive, try turning the weighting down closer to 0.1.

6.2 Trajectory Calibration

There are only two metrics available for trajectory calibration; the weight of the trajectory positions themselves and also the facing angle of the trajectory points. As mentioned, previously, it's unreasonable to compare an angle to a distance so we need to give angles a very low weight and the positions a much higher weight to normalise the comparison.

6.3 Pose vs Trajectory Balancing

Once you are comfortable with your pose and trajectory weightings it's time to start balancing them. This is done via the pose-trajectory slider as shown in Figure 6.4. This can also be thought of as fluidity vs responsiveness.

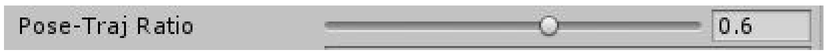

Pose-Traj Ratio 0.6

Figure 6.4 'Pose vs Trajectory Slider'

If the slider leans more on the trajectory side, the animation will be more responsive but will lose some quality / fluidity. Alternatively if the slider leans more on the pose side, animation will be more fluid but you will lose responsiveness. This is a balancing act and the trick is to tweak this until you get the results you desire.

6.4 Multiple Calibrations

When it comes to calibration, one size certainly does not fit all. It is likely that you will need multiple different calibrations for different movement types. For example, strafing motion mode requires a higher weighting on trajectory facing angle than position to prevent the character from turning.

It is possible to add as many calibration sets as you like to the MxMAnimData by choosing the ‘Add New Calibration’ button. Each calibration can be named and the calibration used can be switched at runtime using this name or an integer ID of the calibration. See section 8.1 for more details on switching calibration through code at runtime.

6.4 Iteration

Getting pose-trajectory calibration right is an iterative process that belongs predominantly in the polish phase. It is not recommended to constantly calibrate early in your project as any future change to your pose or trajectory configuration will require re-calibration. Even adding new animations to your library could affect calibration. As a result it is recommended only to calibrate to an extent to get ‘ok’ results. Final calibrations can take place in the polish phase where all your animations are locked in.

Also note that calibration can be difficult at first as it may seem arbitrary. However, as you see the results of your calibration, you will start to notice what is causing specific issues. Below is a list of some common side effects of specific calibration problems.

IMPORTANT: Not all issues are caused by calibration. Ensure that you haven’t accidentally setup animations incorrectly or tagged incorrectly before assuming calibration is the issue. Also ensure your trajectory generator is calibrated correctly as well with an Input Profile.

Character runs off on it’s own despite input - Your pose is too strong, move the slider to the right or reduce the weighting of your pose aspects. Alternatively increase the position weighting of the trajectory. This may also be caused by ‘Favour Current Pose’ setting in the MxMAnimator or other pose favour factors.

Character’s animation is janky - Your trajectory is too strong, move the slider to the left or reduce the weighting of your trajectory aspects.

Character follows the trajectory but is facing the wrong way - You need to increase the weight of your trajectory facing angles or alternatively reduce the weight of your trajectory positions. Use ‘favour’ or ‘require’ tags to differentiate strafing actions and non strafing actions.

The upper body of my character is not fluid but the bottom half is - Consider increasing the weighting of the position and velocity of your stabilizer joint (e.g. hips / neck joint)

My character can go from ‘idle-to-walk’ and ‘idle-to-run’ but it struggles to transition from ‘walk-to-run’ and ‘run-to-walk’ - This is caused by a lack of animation transitions and not having continuity between your walk and run animations. The ideal solution is to provide the system with ‘walk-to-run’ and ‘run-to-walk’ animations. However, by increasing the trajectory weighting and reducing the pose weighting (specifically resultant velocity costing), the cost function will allow MxM to jump the gap in continuity.

My character sometimes changes locomotion speed especially when making tight turns or plants, even though it should stay at the same locomotion speed - First and foremost, check that your trajectory is calibrated properly. For example, ensure that your trajectory speed is high enough to adequately match your animations.

If the trajectory is calibrated correctly, use favour tagging to differentiate different move speeds and request favour tags at runtime based on user input. The MxMDemo scene has a script that does this called ‘LocomotionSpeedRamp.cs”.

On rare occasions this can be caused by ‘Body Velocity' weight being too low or too high.

There are many more examples, but these should help you get started. Understanding calibration comes with time and experience with the system.

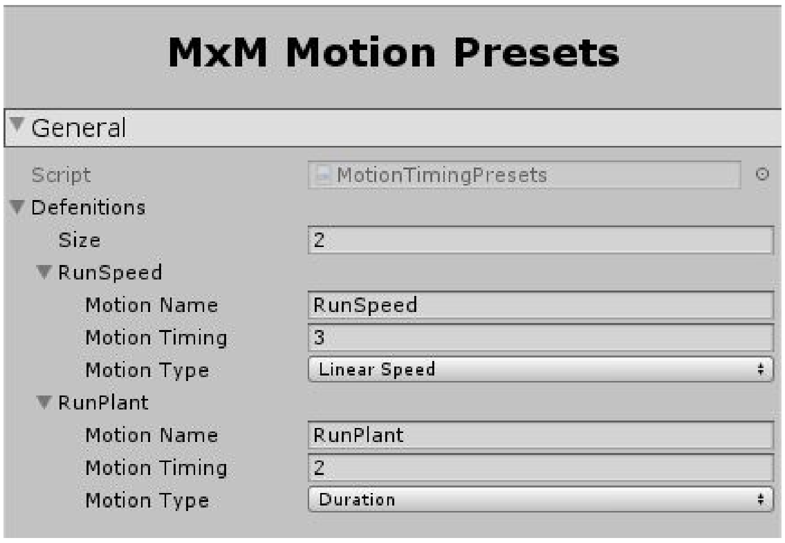
7 Motion Timing Presets

Motion Timing Presets enable the manipulation of your Animation Clip assets during the pre-process phase to better match gameplay. Often purchased or mocap animations don't match gameplay very well. This can be a problem for motion matching in particular as it is explicitly trying to find the best animation to match your gameplay.

With motion timing presets and the speed manipulation tool (in the timeline editor), you can manipulate the speed of parts of your animations without destroying the original animation. However, it's not just a straight playback speed multiplier. It's much smarter than that.

7.1 Creating Presets

First comes the presets themselves. In the 'Motion Timing Presets' foldout of your Pre-processor, click the 'Create New Presets' button. This will generate a new presets data asset which becomes embedded in the pre-processor asset. By double clicking it you can view its inspector. It is essentially a list of data structures which you should populate with common gameplay timing variables. Give them a name, a value and a type. Figure 7.1 shows an example of some typical locomotion timing presets.



MxM Motion Presets

General

Script: MotionTimingPresets

Defenitions

Size: 2

RunSpeed

Motion Name: RunSpeed

Motion Timing: 3

Motion Type: Linear Speed

RunPlant

Motion Name: RunPlant

Motion Timing: 2

Motion Type: Duration

Figure 7.1 'Typical locomotion timing presets'

7.1.1 Motion Types

There are three primary motion types and they describe how a section of animation should be modified based on the value of the variable. The types function as follows:

Playback Speed - This is the most simple type. It just multiplies the animation playback speed by the value of the variable.

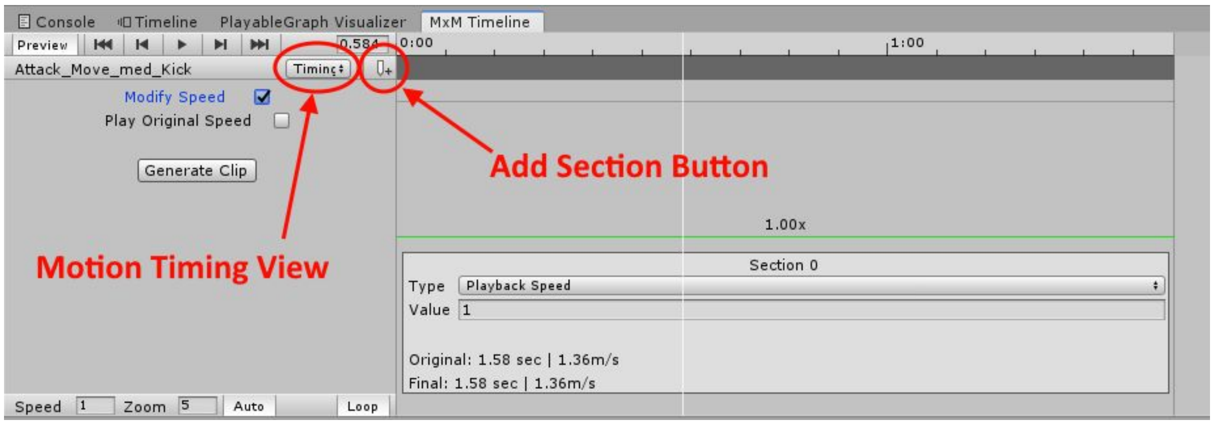
Duration - This motion type allows you to specify a duration of a motion. For example, you may specify a walking plant to take 0.4 seconds. You could then mark-up a section in the timeline editor encapsulating a walking plant and assign this variable to it. It will calculate how much it has to modify the speed of that section to achieve exactly 0.4s.

Linear Speed - This is another complex type which is often used for cycles like walking or jogging. The system will calculate the speed modification to ensure that the average speed of the section matches exactly the value of the variable in m/s. It’s important to note that this is the average linear speed of each frame and does not take into account direction.

7.2 Modifying Animation Speed

Now that you have created your motion timing presets asset and added some presets, it's time to actually modify your animations. Modification can be made in the same timeline window used to add tags or events (Section 3.3). Click the drop down button as shown in Figure 7.2 and select ‘Motion Timing’ to switch to timing mode. To modify animations ensure you check the ‘Modify Clip Speed’ checkbox on the left panel.

By default, every animation has one speed section which cannot be deleted, to make a split in your animation, and simultaneously a new section, scrub the timeline to the point you want to make a split and click the ‘Add Section’ button as shown in Figure 7.2.



Section 0

Type: Playback Speed

Value: 1

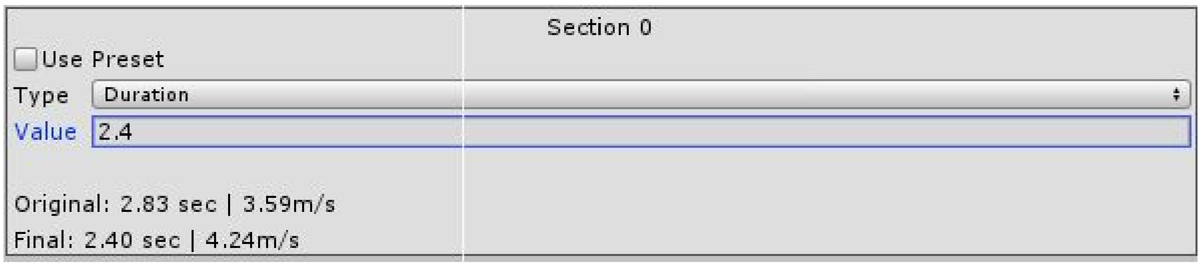
Original: 1.58 sec | 1.36m/s

Final: 1.58 sec | 1.36m/s

Speed 1 Zoom 5

Figure 7.2 ‘Speed manipulation timeline editor’

Each section contains a green line indicating the resulting speed multiplier for the section. You can manually specify custom values for each section in its corresponding data block but it is recommended to use the presets by checking the ‘Use Preset’ checkbox instead. This will allow you to select a variable specified in your ‘Motion Timing Presets’ data for that section.



Section 0

Use Preset

Type: Duration

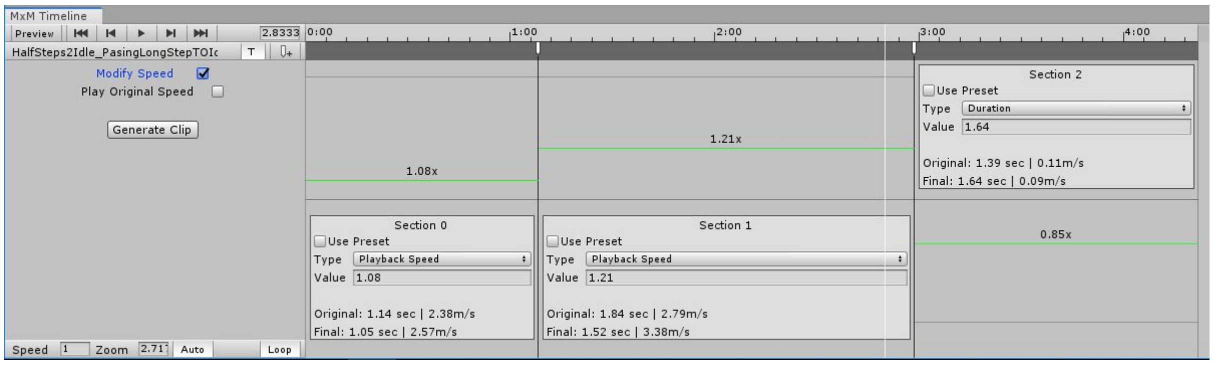
Value: 2.4

Original: 2.83 sec | 3.59m/s

Final: 2.40 sec | 4.24m/s

Figure 7.3 ‘Motion section data block’

The advantage of using ‘Motion Timing Presets’ over manual entry is that you only have to modify the variable in the presets file to apply that modification to all animations where it has been set. This is great for iteration and consistency.



Section 0

Use Preset

Type: Playback Speed

Value: 1.08

Original: 1.14 sec | 2.38m/s

Final: 1.05 sec | 2.57m/s

Section 1

Use Preset

Type: Playback Speed

Value: 1.21

Original: 1.84 sec | 2.79m/s

Final: 1.52 sec | 3.38m/s

Section 2

Use Preset

Type: Duration

Value: 1.64

Original: 1.39 sec | 0.11m/s

Final: 1.64 sec | 0.09m/s

Figure 7.4 ‘Animation with multiple speed sections’

7.3 Reviewing Animation Speed

It is recommended to continually review your animations as you manipulate the speed to ensure that you are getting the results you want. To do this, ensure the ‘Preview’ button is toggled on in the top left corner of the timeline editor. This will spawn the model relating to your pre-process data for real time viewing of the animation pose at the current time in your timeline.

If ‘Modify Speed’ is checked the playback speed will be warped to show what the speed manipulation will look like once generated. To view the original speed, check the ‘Original Speed’ toggle. Note that this second toggle will not prevent the animation’s speed from being manipulated during pre-processing.

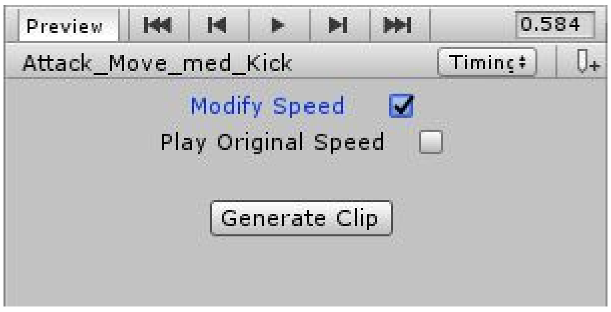


Figure 7.5 ‘Speed manipulation left panel’

Another way to review, is to actually generate this clip from this editor. This will create an animation clip asset which can be used like any other animation in Unity.

There is no need to re-assign the generated clip into your composite, as the pre-processor will automatically use it if it has been generated, either here or during pre-process.

Note: It is recommended to generate all motion clips during the the actual pre-process by checking the ‘Generate Modified Animations’ toggle in the pre-process foldout of the preprocessor inspector.

7.4 Limitations of Speed Manipulation

Speed manipulation is a great tool for massaging your animations to be more precise and responsive. Please note, however, that there is a limit to how much you can speed up or slow down animations before they start to look strange. Keep this in mind when manipulating animation speeds. It’s better to have animations as close as possible before considering speed manipulation in the pre-process stage.

Also note that this tool does not replace good animation in the first instance. It is recommended to use the speed manipulation tool as a way to iterate quickly, and use that knowledge to inform animators to modify the clips from the source so that any strange speed behaviour is removed.

8 Coding With Motion Matching

The ability to code is very important to achieve the best results with motion matching. Do not expect to create custom gameplay with MxM if you cannot code. As stated multiple times throughout the documentation, MxM is an animation system and does not handle gameplay. As a developer it is your responsibility to code your own gameplay and feed the relevant information into MxM.

Note: Visual scripting packages like 'Bolt' that don't require custom integrations work very well with MxM as an alternative to traditional scripting.

This section will cover the basics of coding with MxM. For more advanced coding topics, see Section 11 which explains how to create a custom gameplay controller that interfaces with MxM.

Before diving into details or calling any functions from the MxMAnimator, you will need to obtain a handle to the component. This can be done in the typical way from your script attached on the same object

```
MxMAnimator myMMAnimator = GetComponent<MxmAnimator>();
```

Almost all the settings exposed in the MxMAnimator inspector can, and should, be modified at runtime to achieve the best results. This is done mostly through properties and setter functions defined in the MxMAnimator.

8.1 Changing Basic Settings at Runtime

In order to make a convincing and flexible character, you will want to make changes to the MxMAnimator settings at runtime. Almost all of the settings can be changed at runtime using Property setters. The property setters available are as follows:

Blending times

```
float BlendTime
```

Playback Speed

```
float DesiredPlaybackSpeed
float PlaybackSpeedSmoothRate
float PlaybackSpeed
```

Note: Use 'DesiredPlaybackSpeed to change the playback speed smoothly overtime. Use 'PlaybackSpeed' to change it instantly.

Trajectory Blending

```
bool BlendTrajectory
float TrajectoryBlendWeight
```

Blend Space Settings

```
EBlendSpaceSmoothing DefaultBlendSpaceSmoothType
Vector2 DefaultBlendSpaceSmoothRate
```

Trajectory Warping

```
EAngularErrorWarp AngularErrorWarpType
EAngularErrorWarpMethod AngularErrorWarpMethod
ELongitudinalErrorWarp LongErrorWarpType
float AngularErrorWarpRate
float AngularErrorWarpThreshold
```

Other

```
AvatarMask AnimatorControllerMask
float UpdateRate //UpdateInterval for MxM v2.2 and earlier
```

8.2 MxMAnimator Setter Functions

There are some utility setting functions that allow you to change multiple settings at once in the MxM animator. These functions are as follows.

```
public void SetPlaybackSpeed(float a_speed, bool a_smooth)

public void FavourCurrentPose(bool a_favour, float a_favourMultiplier)

public void SetAngularErrorWarping(EAngularErrorWarp a_type,
EAngularErrorWarpMethod a_method, float a_compensationRate, float
a_distThreshold, float a_angleThreshold)
```

8.3 Switching Between Calibration Sets

It is possible to have multiple calibration sets for each MxMAnimData. You may choose to have different calibrations for different situations such as different stances. In such cases you need to make multiple calibrations for the MxMAnimData and it is recommended you name them appropriately.

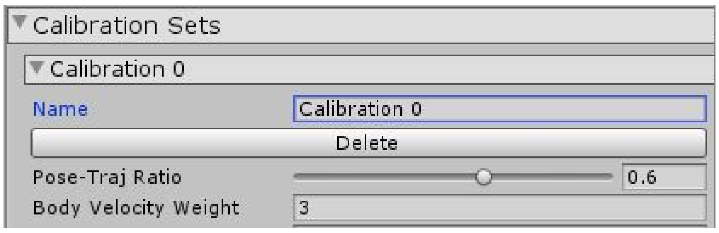

Calibration Sets
Calibration 0
Name: Calibration 0
Delete
Pose-Traj Ratio: 0.6
Body Velocity Weight: 3

Figure 8.1 'Calibration set naming'

To switch to a different calibration, call one of the following functions:

```
public void SetCalibrationData(string a_calibDataName)

public void SetCalibrationData(int a_calibDataId)
```

8.4 Switching Between MxM and Mecanim

You can fade/blend between MxM and mecanim by calling one of the following functions

```
public void BlendInController(float a_fadeRate)

public void BlendOutController(float a_fadeRate)
```

If you wish to use the mecanim animator controller for upper body layered animation, the avatar mask for the controller can be set in the inspector or at runtime through the property

```
public AvatarMask AnimatorControllerMask
```

8.5 Switching Between MxMAnimData Sets

In some cases, it may be appropriate to use multiple MxMAnimData assets to control your animation. For example, you may split up locomotion animations into one AnimData and and climbing into another. This is necessary because each of these motion types requires different bones to be matched.

```
public void SwapAnimData(int a_animDataId, int a_startPoseId)
```

MxMAnimData assets are ID'd in the order that they are placed in the MxMAnimator inspector.

One of the downfalls of switching animation data sets is that it is impossible to match animations between them due to the potential disparity between their trajectory and pose configurations. To overcome this, first call an event that can be used as a transition animation (e.g. mounting a wall animation event) and then immediately queue a swap with the following function

```
public void QueueAnimDataSwap(int a_animDataId, int a_startPoseId)
```

This will cause the animator to wait until the current event has concluded before swapping the animation data.

8.6 Managing Tags

8.6.1 Requiring Tags

Tags are similar to bool variables that you would use with mecanim (except faster). Just like mecanim you have to tell MxM what tags are required. One example use case of this is to notify the animator that your player is crouching. When your player presses the button to crouch, you would call the following function, passing in the name of the tag you set for crouching in the pre-processor.

```
public void SetRequiredTag(string a_tagName)
```

If you know the Id of the tag you can also use one of the following functions. They are much more efficient as they do not have to perform string comparisons and cannot allocate garbage.

```
public void SetRequiredTag(int a_tagId)

public void SetRequiredTags(ETags a_tags)
```

When a tag is set to be required, MxM will only ever play animations that have been tagged with the required tag or tags. You can require multiple tags at once if appropriate and any candidate pose must have all the required tags. To add new required tags to your current required tag list, call one of the following:

```
public void AddRequiredTag(string a_tagName)

public void AddRequiredTag(int a_tagId)

public void AddRequiredTags(ETags a_tags)
```

To remove a tag from the required tag list, call one of the following functions:

```
public void RemoveRequiredTag(string a_tagName)
public void RemoveRequiredTag(int a_tagId)

public void RemoveRequiredTags(ETags a_tags)
```

You can clear all required tags with:

```
public void ClearRequiredTags()
```

8.6.2 Favouring Tags

Depending on your game, there may be times where some tags could be favoured rather than required. A favoured tag will have its cost reduced making it a more likely candidate for selection. To set the favoured tag/s call one of the following functions:

```
public void SetFavourTag(string a_tagName, float a_favour)

public void SetFavourTag(int a_tagId, float a_favour)

public void SetFavourTags(ETags a_tag, float a_favour)
```

Just like with required tags, you can choose to favour more tags than one, but a pose only needs to have one of the favoured tags to obtain the benefit. To add tags to your current favouring list call call one of the following:

```
public void AddFavourTag(string a_tagName)

public void AddFavourTag(int a_tagId)

public void AddFavourTags(ETags a_tags)
```

The favour factor specifies how much a tagged pose will be favoured. It is a multiplier to the cost of the pose so numbers less than 1 will provide favour while any number above 1 will make it a less favourable tag.

It is also possible to change the favour multiplier without modifying the favour tags:

```
public void SetFavourMultiplier(float a_favour)
```

To remove favoured tags simply call one of the following:

```
public void RemoveFavourTag(string a_tagName)

public void RemoveFavourTag(int a_tagId)

public void RemoveFavourTags(ETags a_tags)
```

Finally, to clear all favoured tags, call:

```
public void ClearFavourTags()
```

All tags requirements and favouring can be cleared with the following call:

```
public void ClearAllTags()
```

Note: Favour tags and required tags are not compatible. At any given time you must choose between one or the other. It is far more common to use required tags exclusively.

8.6.3 User Tags

Unlike ‘require’ and ‘favour’ tags, user tags are not set. Rather they can be queried from the MxMAnimator to determine if the currently playing animation pose has specific tags. The user can then run their own logic based on what tags are active.

To query the MxMAnimator for currently active user tags, use one of the following functions.

```
public EUserTags QueryUserTags()
public bool QueryUserTags(EUserTags a_queryTags)
public bool QueryUserTag(int a_queryTagId)
public bool QueryUserTag(string a_queryTagName)
```

Note that the enumeration ‘EUserTags’ is a flag enum, meaning that it can hold the data for multiple tags at once. The fastest method of querying tags is to simply request the current tags ‘EUserTags’ value using the first function above.

8.6.4 Caching Tag Handles

For performance reasons it is never a good idea to pass strings when working with tags. Not only does it allocate garbage, but MxM has to search for the tag based on the string, every time it is called. This is a massive overhead to perform something that is quite simple. To avoid this, users should query and cache tag handles at load time. Then at runtime, use the ETags methods instead of the string methods.

Require, Favour and User tags handles can be cached at load time via the following method.

```
ETags MxMAnimator.CurrentAnimData.TagFromName(string a_tagName);

ETags MxMAnimator.CurrentAnimData.FavourTagFromName(string a_favourTagName);

EUserTags MxMAnimator.CurrentAnimData.UserTagFromName(string a_userTagName);
```

8.7 Triggering Events

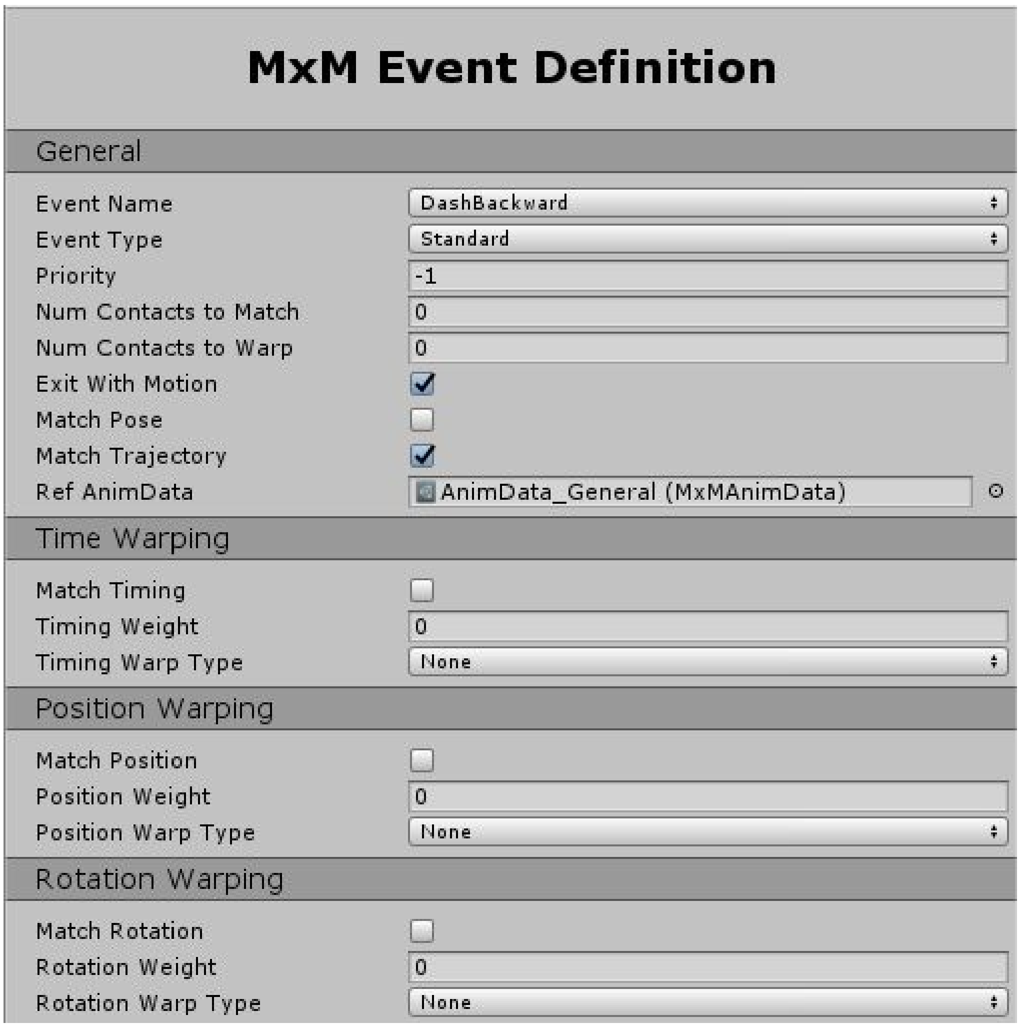
Much like tags, you have to tell the MxMAnimator when you want it to trigger an animation event. This is not like events set in Unity animation clips, but rather events in the animations that you specify and relate to specific action. See the events section 3.3.1 for more details.

In short, events are actions like a sword hit or a vault. When your gameplay decides that it's time to perform one of these actions you need to call one of the BeginEvent functions from the MxMAnimator.

From version 1.3b+, any kind of event can be called using a single function as follows:

```
public void BeginEvent(EventDefenition a_eventDefenition)
```

This BeginEvent function takes a reference to a special asset which can be created in the project view by right clicking and choosing 'Create/MxM/Core/Modules/Event Definition'. The inspector for an event definition is shown in Figure 8.2


MxM Event Definition

General

Event Name	DashBackward
Event Type	Standard
Priority	-1
Num Contacts to Match	0
Num Contacts to Warp	0
Exit With Motion	✓
Match Pose	☐
Match Trajectory	✓
Ref AnimData	AnimData_General (MxMAnimData)

Time Warping

Match Timing	☐
Timing Weight	0
Timing Warp Type	None

Position Warping

Match Position	☐
Position Weight	0
Position Warp Type	None

Rotation Warping

Match Rotation	☐
Rotation Weight	0
Rotation Warp Type	None

Figure 8.2 'Event Definition Inspector'

This asset object lets you pre-define a lot of the meta-data to get an event running so all you have to do at runtime is fill it with world contact points and pass it to the MxMAnimator through the Begin Event function.

The first thing to do after creating this asset is to drag in a reference of your MxMAnimData into the 'Ref AnimData' field. This will allow you to choose events names from the anim data without having to figure out the specific event Ids number.

All other parameters are as follows:

- **Event Id / Event Name -** The event Id for this event (or event name if you have an MxMAnimData slotted in the 'Ref AnimData' field.
- **Event Type** - What kind of event to play?
    - *Standard* - a normal 1 shot event
    - *Loop* - an event that loops over and over until manually exited through code
    - *LoopSequence* - an event that plays in a sequence from the windup phase but loops the action and follow through phase. The recovery phase is played on termination.

- **Priority -** Default priority of -1 overrides any event no matter what. Otherwise, the event will only play if its priority is greater than any current event.
- **Num Contacts to Match** - The number of contacts to match when calculating the best event and pose to jump to.
- **Num Contacts to Warp -** The number of contacts to warp animations between to get precise contacts.
- **Exit with Motion -** whether the event should allow the user to exit the recovery phase of the event with motion.
- **Match Pose -** Whether to match/score the current pose when considering this event
- **Match Trajectory -** Whether to match/score the current trajectory when considering this event
- **Match Timing -** Whether to match/score the timing when considering events
- **Timing Weight -** The scoring weight for timing
- **Timing Warp Type** - The type of warping to use for timing *(Note: Linear, Simple Dynamic and Dynamic produce the same result for timing)*
    - None - no time warping
    - Linear / Simple Dynamic / Dynamic - time warping
- **Match Position -** Whether to match/score contact positions when considering events
- **Position Weight -** The scoring weight for position.
- **Position Warping Type -** The type of warping to use for position warping
    - None - no warping
    - Linear - linear warping over time
    - Dynamic - warps by scaling root motion
- **Match Rotation -** Whether to match/score contact rotations when considering events
- **Rotation Weight -** The scoring weight for rotation
- **Rotation Warping Type** - The type of warping to use for rotation warping
    - None - no warping
    - Linear - linear rotation over time
    - Dynamic - warps by scaling rotation

You will need to provide your gameplay code with references to these assets so you can use them at runtime.

During runtime, when you trigger an event for whatever reason, you can pass the Event Definition to the MxMAnimator. If the event requires no environmental contacts, there is no need to do anything before calling begin event.

If your event has contact points or a desired timing offset / delay, you must first clear all existing contacts from the definition with:

```
public void EventDefinition.ClearContacts()
```

Once the event definition is cleared you can add contact points to it with any of the following functions. Note that all contacts are taken in as world positions and rotations.

```
public void AddEventContact(EventContact a_contact)

public void AddEventContact(Vector3 a_position, float a_rotationY)

public void AddEventContact(Vector3 a_position, Quaternion a_rotation)

public void AddEventContact(Transform a_contactTransform)

public void AddEventContacts(Transform[] a_contactTransforms)

public void AddEventContacts(List<Transform> a_contactTransforms)
```

To specify a desired timing for your event, simply set the DesiredDelay property on the EventDefinition.

```
public float DesiredDelay;
```

Once you have added all your contacts, the Event Definition can be passed to the MxMAniamtor through the BeginEvent function.

If you would like to manually stop an event via code before it is finished, you can call the following function, and normal motion matching will resume:

```
public void ForceExitEvent();
```

8.7.1 Updating Desired Contact Points

For more complex systems, it may be beneficial to first trigger an event and then update it's contact points at specific times during the event.

For example, you may want to know where a jump is going to end up (without animation warping), and then raycast down from above that point to find where it should land on a surface. However, since motion matching could pick different animations, you may not know where it will land before it is triggered. Therefore, it needs to be updated after it is triggered.

To do this, first clear the contacts on your current event definition and then add dummy contacts with a count equal to the number of contacts you desire.

```
public void AddDummyContacts(int a_dummyContactCount);
```

The event can then be triggered as normal without any problems. Immediately after the event the actual contact point of the animation (without warping) can be extracted from the MxMAnimator through the following property

```
public EventContact MxMAnimator.NextEventContact_Actual_World;
```

With this information, your gameplay logic can use raycasts to get the desired contact point and then update it with one of the following calls:

```
public void ModifyDesiredEventContact(EventContact a_desiredContact)

public void MxMAnimator.ModifyDesiredEventContactPosition(Vector3
a_position);

public void ModifyDesiredEventContactRotation(float a_desiredRotationY)
```

This method could be combined with event contact callbacks to update desired contacts for multi-contact events at different times in the event.

Note: If you do not plan to update the contact point immediately after the event is called, it is recommended that warping be turned off in your event definition and only turned on through code with the following properties, after the contacts have been updated.

```
public EEventWarpType MxMAnimator.WarpType { get; set; }

public EEventWarpType MxMAnimator.RotWarpType { get; set; }

public EEventWarpType MxMAnimator.TimeWarpType { get; set; }
```

8.8 Pausing the MxMAnimator

The MxMAnimator can be paused or un-paused with the following functions. Their use is self explanatory.

```
public void TogglePause()

public void Pause()

public void UnPause()
```

9 Animation Layer System

Motion matching is at its best with full body animation. Using layers somewhat undermines the point of high quality organic animations as they cannot be guaranteed to fit in with the organic motion. Regardless, ‘Motion Matching for Unity’ supports animation layers with AvatarMasks for those cases where you don’t have a choice.

The motion matching system takes up the first layer, while the Animator Controller takes up the second (if you are using one). All other layers are blended over the top of it.

9.1 Creating Layers

To create a custom layer call one of the following functions which either takes an AnimationClip directly or a custom playable.

```
public int AddLayer(AnimationClip a_clip, float a_weight,
    bool a_additive, AvatarMask a_mask)

public int AddLayer(Playable a_playable, float a_weight,
    bool a_additive, AvatarMask a_mask)
```

This function returns an int representing the id of the layer you just created. You will want to cache (keep) this id so you can manage the layer later.

If you are familiar with Unity’s playable graphs, you will understand that any playable node created needs to be created within the playable graph. To Get a reference to the MxM playable graph call:

```
public PlayableGraph GetPlayableGraph()
```

9.2 Manipulating Layers

To blend layers in or out, call one of the following functions, passing in the id of the layer which you cached when you created the layer.

```
public void BlendInLayer(int a_layerId, float a_fadeRate,
     Float a_targetWeight = 1f)

public void BlendOutLayer(int a_layerId, float a_fadeRate,
     float a_targetWeight = 0f)
```

These functions use coroutines to fade layers in and out over time so you can fire and forget.

9.3 Setting Layers

There’s no need to create a new layer every time you want a layered animation. I recommend only one layer for each AvatarMask for simplicity sake. If you want to change a layer that already exists, you can simply call the SetLayer functions:

```
public int SetLayer(int a_layerId, AnimationClip a_clip, float a_weight,
    bool a_additive, AvatarMask a_mask)

public int SetLayer(int a_layerId, Playable a_playable, float a_weight,
    bool a_additive, AvatarMask a_mask)
```

9.4 Deleting Layers

To delete a layer that you no longer need call the following function, passing in the layer id you obtained when you created the layer.

```
public bool RemoveLayer(int a_layerId, bool a_destroyPlayable)
```

You can also delete all layers by calling:

```
public void RemoveAllLayers(bool a_destroyPlayable)
```

9.5 Layer Setters

The following are self explanatory setter functions for layer settings and can be used to change specific layer parameters.

```
public void SetLayerWeight(int a_layerId, float a_weight)

public void SetControllerLayerWeight(float a_weight)

public void SetLayerAdditive(uint a_layerId, bool a_additive)

public void ResetLayerWeights()

public void ResetPlayableGraph()
```

9.5 Layer References

Layers in MxM are their own objects (MxMLayer). To directly access a layer object call the following function, passing the layer id handle provided when creating the layer.

```
public MxMLayer GetLayer(int a_layerId)
```

The following properties can be called to access the current state of the layer:

```
public bool IsDone
public float TimeRemaining
public float Duration
public AnimationClip PrimaryClip
public AvatarMask Mask
public bool Additive
public float PrimaryClipTime
public float TransitionRate
```

10 Creating Your Own Trajectory Generator

Unlike a typical controller where its goal is to run game logic for the current frame, trajectory generators that work with motion matching require something a little extra. They need to run that same logic for the future as well. Each frame you will have to calculate a number of steps into the future, up to the last trajectory point. The steps don't need to be your target frame rate though, usually 10 steps is sufficient for a 1 second trajectory.

Controllers also need to record the past portion of the trajectory. This is typically the same for every trajectory generator, so it's not a wheel that needs to be re-invented.

In essence, the job of the trajectory generator is to feed the MxMAnimator the desired motion of the player, based on their inputs. There are two main ways to create a custom controller that works with motion matching. These methods are described below. Choose the one that best suits you.

10.1 MxMTrajectoryGeneratorBase

This is the recommended way to create a custom trajectory generator for use with MxM. It is a base class that automatically handles past trajectory recording and creates the containers that must be filled with future trajectory predictions. It also automatically interfaces with the MxMAnimator so all you really have to do is run your logic and fill the trajectory containers.

The trajectory generator 'MxMTrajectoryGenerator', that comes with MxM inherits from this base class.

There are two serialized variables in the MxMTrajectoryGeneratorBase which need to be set and use:

```
[SerializeField] protected float p_recordingFrequency = 0f; //Time
between recording passed trajectory points

[SerializeField] protected float p_sampleRate = 20f; //The number of
samples per second for your trajectory
```

The sample rate is the number of samples taken per second for predicting your future trajectory. The trajectory generator base will use this to calculate the total number of samples required and the time between these samples and store these values as:

```
protected int p_timeStep; // The time between trajectory samples
protected int p_trajectoryIterations; //The number of iterations
required for the target sample rate and trajectory.
```

When you run your future prediction loop (see MxMTrajectoryGenerator for an example), you will run the loop `p_trajectoryIterations` times with a time delta of `p_timeStep`

During this loop you must fill the trajectory containers, held in the MxMTrajectoryGeneratorBase with your trajectory points. The containers are as follows:

```
protected NativeArray<float3> p_trajPositions;
protected NativeArray<float> p_trajFacingAngle;
```

You do not need to worry about extracting your trajectory points specified in the pre-processor, the MxMTrajectoryGeneratorBase will automatically interface with your MxMAnimator and do this for you at runtime.

10.2 IMxMTrajectory

If you want to create your own trajectory prediction model from scratch that interfaces with the MxMAnimator then it has to implement the ITrajectoryGenerator interface. This is the minimum requirement.

This is for advanced users so I won't go into detail on how to achieve this. See 'MxMTrajectoryBase' and 'MxMTrajectoryGenerator' for examples of how to implement this interface.

10.3 Handling Root Motion

WARNING: From MxM v2.2 HandleRootMotion now takes root deltas instead of absolutes.

If you choose to handle root motion yourself then you first need to make sure that 'Root Motion' is set to 'Root Motion Applicator' in the MxMAnimator component of your character. You will then need to implement the IMxMRootMotion interface in a new script component or by combining it with your trajectory generator.

Within your IMxMRootMotion class you need to implement the following functions:

```
void HandleRootMotion(Vector3 a_rootPosition, Quaternion a_rootRotation,
                      Vector3 a_warp, Quaternion a_warpRot, float a_deltaTime);
```

This is very similar to how you would normally handle root motion. The system provides a root position and a root rotation from the animator and that is applied to the character. However, with MxM's event system warping, there are additional variables for warping i.e. `Vector3 a_warp Quaternion a_rotWarp`.

If you are handling root motion yourself and still want to use MxM's warping for events then you need to add the `a_warp` to your positional movement. In the simplest sense this means adding `a_rootPosition` and `a_warp` together to get the final position. For rotational warping you will need to multiply `a_rootRotation` with `a_warpRot`.

For an example implementation of IMxMRootMotion, please see the ‘MxMRootMotionApplicator’. This script applies the root motion to a movement character controller instead of the object transform and also adds gravity.

The RootMotionApplicatorTemplate script is a template that can be used to create your own.

Not Using Root Motion?
If you choose not to use root motion to move your character, you may still need a root motion applicator for Angular Error Warping and also for Events (if you want contacts to work). Therefore, it is still recommended to have a root motion applicator.

To handle angular error warping, the following function needs to be implemented. It should just rotate the character by the past `a_warpRot` value.

```
void HandleAngularErrorWarping(Quaternion a_warpRot)
```

Other functions that must be implemented
The following functions also need to be implemented. See RootMotionApplicatorTemplat.cs script for more details

```
void SetPosition(Vector3 a_position); //Set position of character
void SetRotation(Quaternion a_rotation); //Set rotation of character
void SetPositionAndRotation(Vector3 a_position, Quaternion a_rotation);
```

10.4 Decouple Controllers

Decouple controllers are a way to decouple a character's model / animation from the movement controller, allowing it to stray only slightly from the character's actual position. This is one method to help reduce foot sliding and is similar to the technique used in 'For Honor'.

When using decouple controllers, the character movement is controlled by motion logic rather than root motion. However, the root motion still runs on the model which is then clamped within a certain threshold of the controller.

MxM ships with a script to handle decoupling and an alternative root motion applicator, specifically for decoupling. These scripts include:

- **MxMAnimationDecoupler.cs**
    - This component performs the actual decoupling logic. It does not perform any movement logic. It also has a number of settings for 'Root Motion' blending to enable a hybrid 'root motion / movement logic' approach.
    - This component should be placed at the top level of the character hierarchy, alongside the movement controller.

- **MxMDecoupleMotionApplicator.cs**
    - This component is an alternative IMxMRootMotion component that should be used with decouple controllers instead of 'MxMRootMotionApplicator'

- **ExampleDecoupleMovementControl.cs**
    - This component is an example only, showing how movement logic can be applied to a decouple controller.

10.4.1 Character Hierarchy

The object hierarchy for Decouple Controllers is very different to the normal setup. Essentially the top most game has the controller itself and the model is a child of that GameObject as shown in Figure 10.1.

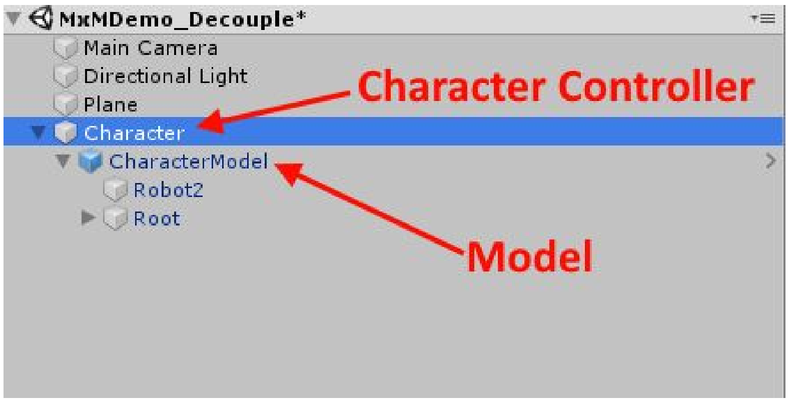


Figure 10.1 Decouple Controller Hierarchy

Character Controller Components:
The character controller game object should have the following scripts:

- MxMAnimationDecoupler
- -- Your custom movement logic component --
- -- Your character controller of choice --
- Generic Controller Wrapper for your controller (if not using a custom IMxMRootMotion Component)

Model Components:

- Animator
- MxMDecoupleMotionApplicator (or custom IMxMRootMotion component)
- MxMTrajectoryGenerator (or custom IMxMTrajectory component)
- MxMAnimator

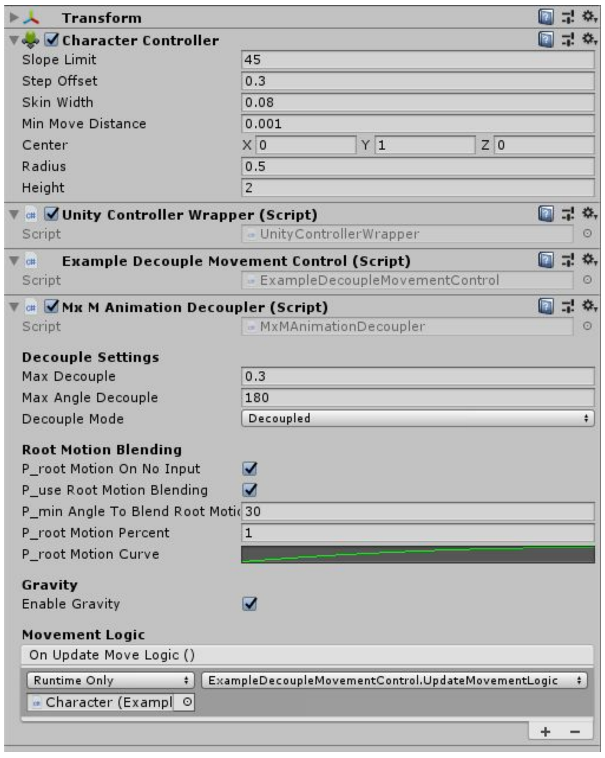

Transform
Character Controller
Slope Limit 45
Step Offset 0.3
Skin Width 0.08
Min Move Distance 0.001
Center X 0 Y 1 Z 0
Radius 0.5
Height 2
Unity Controller Wrapper (Script)
Script UnityControllerWrapper
Example Decouple Movement Control (Script)
Script ExampleDecoupleMovementControl
Mx M Animation Decoupler (Script)
Script MxMAnimationDecoupler
Decouple Settings
Max Decouple 0.3
Max Angle Decouple 180
Decouple Mode Decoupled
Root Motion Blending
P_root Motion On No Input ✓
P_use Root Motion Blending ✓
P_min Angle To Blend Root Moti 30
P_root Motion Percent 1
P_root Motion Curve
Gravity
Enable Gravity ✓
Movement Logic
On Update Move Logic ()
Runtime Only ExampleDecoupleMovementControl.UpdateMovementLogic
Character (Exampl

Figure 10.2 ‘Typical Decouple Controller Setup’

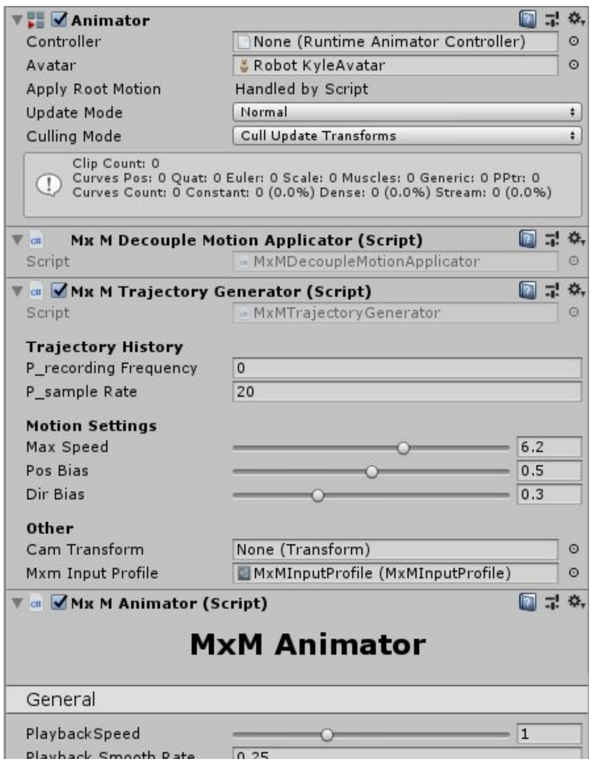
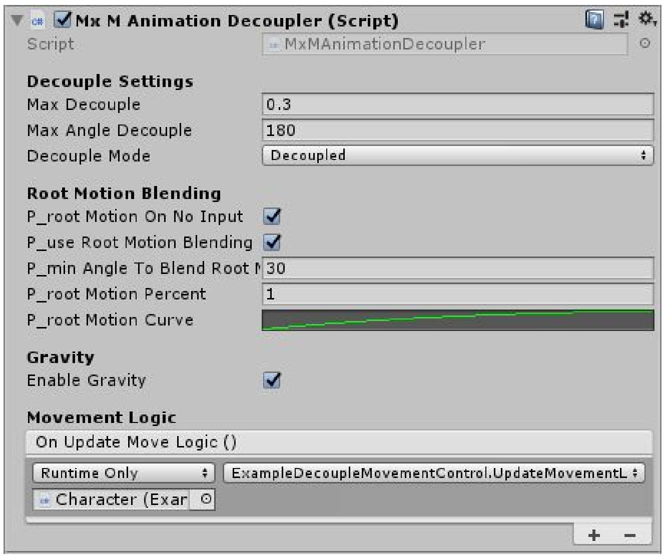

Animator
Controller None (Runtime Animator Controller)
Avatar Robot KyleAvatar
Apply Root Motion Handled by Script
Update Mode Normal
Culling Mode Cull Update Transforms
Clip Count: 0
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0 Generic: 0 PPtr: 0
Curves Count: 0 Constant: 0 (0.0%) Dense: 0 (0.0%) Stream: 0 (0.0%)
Mx M Decouple Motion Applicator (Script)
Script MxMDecoupleMotionApplicator
Mx M Trajectory Generator (Script)
Script MxMTrajectoryGenerator
Trajectory History
P_recording Frequency 0
P_sample Rate 20
Motion Settings
Max Speed 6.2
Pos Bias 0.5
Dir Bias 0.3
Other
Cam Transform None (Transform)
Mxm Input Profile MxMInputProfile (MxMInputProfile)
Mx M Animator (Script)
MxM Animator
General
PlaybackSpeed 1
Playback Smooth Rate 0.25

Figure 10.3 ‘Typical Decouple Model Setup’

10.4.2 Animation Decoupler Options

The MxMAnimationDecoupler inspector (Figure 10.4) has a number of options that are important to understand for good decoupling.


Mx M Animation Decoupler (Script)
Script: MxMAnimationDecoupler

Decouple Settings
Max Decouple: 0.3
Max Angle Decouple: 180
Decouple Mode: Decoupled

Root Motion Blending
P_root Motion On No Input: ✓
P_use Root Motion Blending: ✓
P_min Angle To Blend Root M: 30
P_root Motion Percent: 1
P_root Motion Curve

Gravity
Enable Gravity: ✓

Movement Logic
On Update Move Logic ()
Runtime Only | ExampleDecoupleMovementControl.UpdateMovementL
Character (Exar

Figure 10.3 'Animation Decoupler Options'

Decouple Settings
The following settings control basic decoupling:

- Max Decouple - The maximum distance the model can stray from the controller
- Max Angle Decouple - The maximum angle the model can rotate away from the controller angle (typically leave this at 180 and allow root motion to control the rotation)
- Decouple Mode -
    - Decoupled - allows decoupled position and rotation
    - Lock Rotation - allows decoupled position but locked rotation
    - Lock Position - allows decoupled rotation but locked position
    - Lock All - disables decoupling all together.

Root Motion Blending
Sometimes it is beneficial to blend between custom movement logic (controlled by code) and root motion (controlled by animation) to achieve a nice hybrid. If done correctly this can provide the best of both worlds.

Essentially this system allows automatic blending in of root motion when the character model is not facing the same direction that the controller is moving. This stops the character sliding backward during complex plants and turns. Additionally, root motion can be turned on when there is no user input, allowing for realistic stops.

Root motion blending settings are explained as follows:
- Root Motion On No Input - If checked, root motion will turn on when there is no player input allowing for realistic stops without footsliding.
- Use Root Motion Blending - Enables root motion blending
- Min Angle to Blend Root Motion - When the angle between the character facing direction and the controller velocity direction is greater than this angle, root motion will be blended in with a falloff.

- Root Motion Percent - The maximum percentage of root motion that is allowed to be used (0 - 1)
- Root Motion Curve - A curve to control the falloff of root motion blending between the min angle and 180 degrees.

Gravity

- Enable Gravity - enables gravity for the decouple controller

Movement Logic

In order for movement logic to work correctly with decoupling, it must occur at a very specific time. Use the OnUpdateMoveLogic callback here to run your movement logic update.

11 Creating a Custom Playable Graph

If you desire to integrate motion matching in a custom playable graph, it can be done by checking the DIYPlayableGraph toggle in the MxMAnimator. By checking this toggle, the MxMAnimator will not create its own graph. Instead it will wait until your code asks MxM for a playable to connect into your custom graph.

This is a feature for advanced users only and this guide will not go into detail on how to create a custom playable graph.

From your custom playable graph code, you need to call the following function from the MxMAnimator component.

```
public ScriptPlayable<MotionMatchingPlayable>
CreateMotionMatchingPlayable(PlayableGraph a_playableGraph)
```

This will return a motion matching playable that you can plug into your playable graph anywhere you like. MxM will then continue to manage it’s portion of your playable graph but you can control when it is blended in and out.

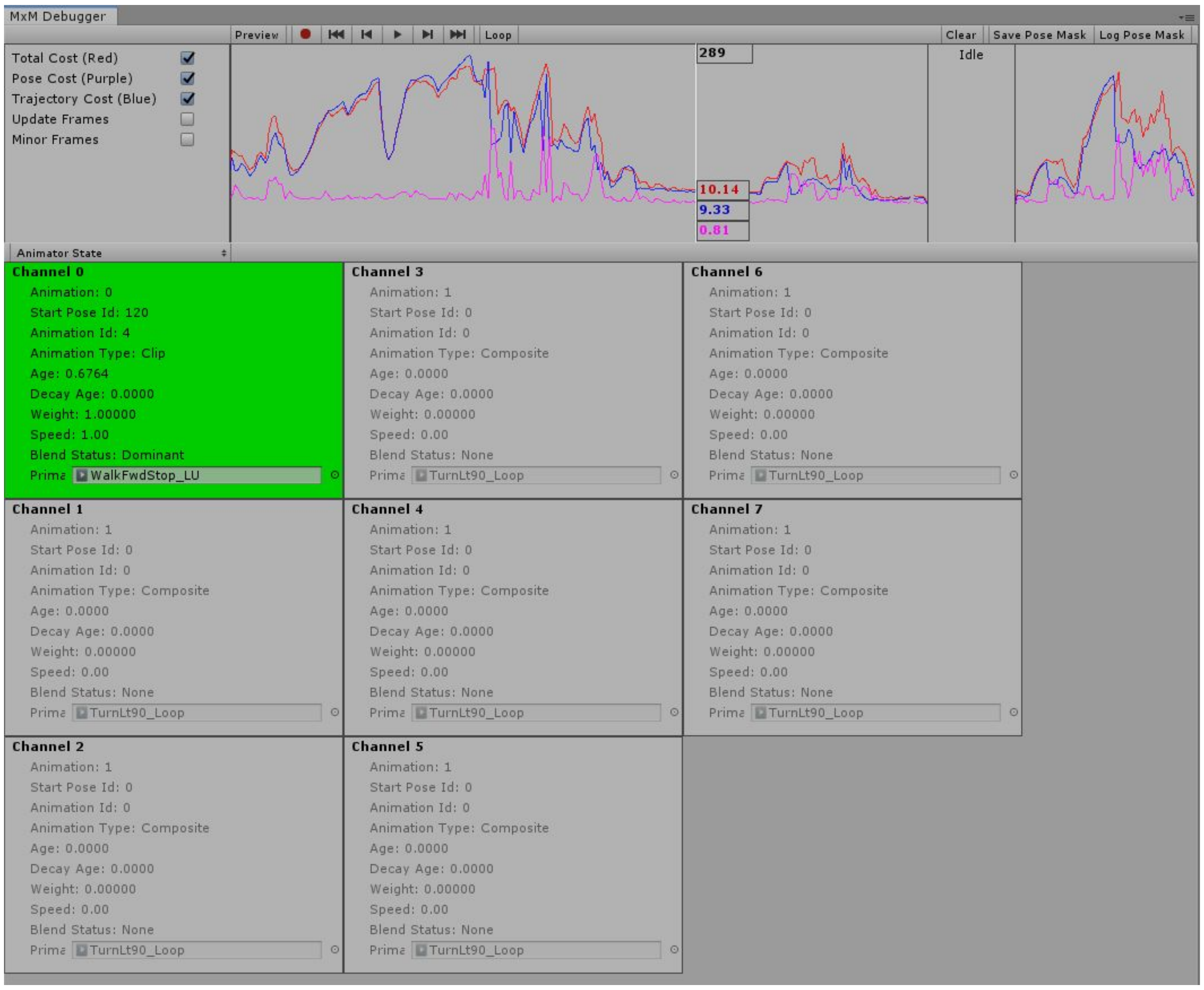
12 Debugging Window

Aside from the debugging features built into the MxMAnimator inspector, there is a powerful debugging window that allows for recording and replay or motion matched animation. This debug window allows you to see any data internal to the MxMAnimator including the results of the cost function.

The MxMDebugger window can be opened through the menu toolbar under *Window/MxM/Debugger*. Alternatively it can be opened directly from the MxMAnimator inspector ‘Debug foldout’.

<u>Note:</u> that the window will not display anything unless in playmode. If playmode is active and the debug window is not displaying anything. Simply select the game object with the MxMAnimator component attached.

The general interface for the debugging window is shown in Figure 12.1.



Total Cost (Red) ✓
Pose Cost (Purple) ✓
Trajectory Cost (Blue) ✓
Update Frames ☐
Minor Frames ☐

Channel 0
Animation: 0
Start Pose Id: 120
Animation Id: 4
Animation Type: Clip
Age: 0.6764
Decay Age: 0.0000
Weight: 1.00000
Speed: 1.00
Blend Status: Dominant
Prima WalkFwdStop_LU

Channel 1
Animation: 1
Start Pose Id: 0
Animation Id: 0
Animation Type: Composite
Age: 0.0000
Decay Age: 0.0000
Weight: 0.00000
Speed: 0.00
Blend Status: None
Prima TurnLt90_Loop

Channel 2
Animation: 1
Start Pose Id: 0
Animation Id: 0
Animation Type: Composite
Age: 0.0000
Decay Age: 0.0000
Weight: 0.00000
Speed: 0.00
Blend Status: None
Prima TurnLt90_Loop

Channel 3
Animation: 1
Start Pose Id: 0
Animation Id: 0
Animation Type: Composite
Age: 0.0000
Decay Age: 0.0000
Weight: 0.00000
Speed: 0.00
Blend Status: None
Prima TurnLt90_Loop

Channel 4
Animation: 1
Start Pose Id: 0
Animation Id: 0
Animation Type: Composite
Age: 0.0000
Decay Age: 0.0000
Weight: 0.00000
Speed: 0.00
Blend Status: None
Prima TurnLt90_Loop

Channel 5
Animation: 1
Start Pose Id: 0
Animation Id: 0
Animation Type: Composite
Age: 0.0000
Decay Age: 0.0000
Weight: 0.00000
Speed: 0.00
Blend Status: None
Prima TurnLt90_Loop

Channel 6
Animation: 1
Start Pose Id: 0
Animation Id: 0
Animation Type: Composite
Age: 0.0000
Decay Age: 0.0000
Weight: 0.00000
Speed: 0.00
Blend Status: None
Prima TurnLt90_Loop

Channel 7
Animation: 1
Start Pose Id: 0
Animation Id: 0
Animation Type: Composite
Age: 0.0000
Decay Age: 0.0000
Weight: 0.00000
Speed: 0.00
Blend Status: None
Prima TurnLt90_Loop

Figure 12.1 ‘MxMDebugger Window’

12.1 Debug Views

The debugging window consists of several views that hold data about the recorded motion matched animation.

12.1.1 Graph View

The top section is the timeline displays several graph lines showing the cost minima for the motion matching algorithm. The graphs can be toggled using the checkboxes on the left most panel. Graph lines are as follows:

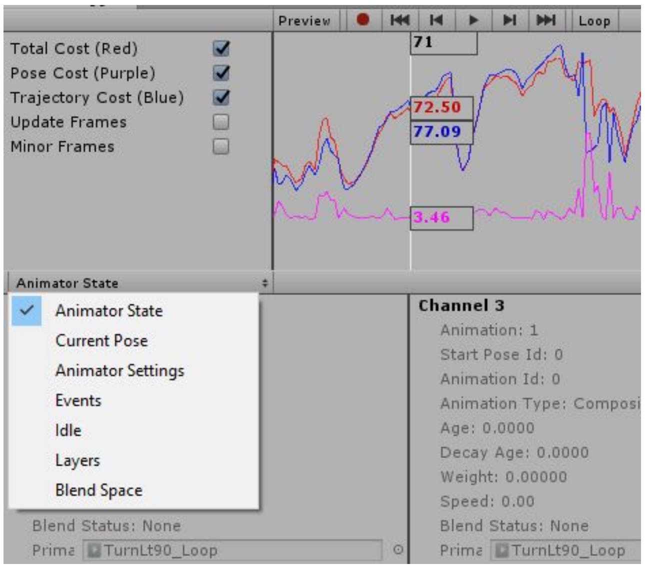
- Red - Total cost (i.e. trajectory + pose)
- Blue - Trajectory Cost
- Purple - Pose Cost

Cost is essentially how MxM chooses the animation pose to jump to. The values displayed here are the minima costs for the chosen pose out of every pose in the animation database. This view can be very useful to assist with calibration and identify anomalies.

Note: The graph shows recorded animation data in a fixed length circular buffer.

12.1.1 Data View

The data view is the bottom most section of the debug window. It contains data that is contextual to the playhead frame position on the graph view. The data displayed is also dependent on the current mode which is chosen from the popup in the middle menu bar as shown in Figure 12.2.



```
Channel 3
    Animation: 1
    Start Pose Id: 0
    Animation Id: 0
    Animation Type: Composi
    Age: 0.0000
    Decay Age: 0.0000
    Weight: 0.00000
    Speed: 0.00
    Blend Status: None
    Prima  TurnLt90_Loop
```

Figure 12.2 'Debug Data View Modes'

The data displayed for each mode is as follows:

- **Animator State** - Displays the state of each Animation channel on the motion matching mixer. Here you can see what animations are active, their blend weights and other useful parameters.
- **Current Pose** - Displays all details on the current pose of the character at the playhead frame position.
- **Animator Settings** - Displays the settings of the MxMAnimator at the playhead frame. These are the same settings set in the inspector. However, if the user changes settings at runtime, those changes will show up here.
- **Events** - Displays any information relating to MxM's event system.
- **Idle** - Displays all information relating to MxM's idle system.
- **Layers** - Displays all information relating to MxM's layer system
- **Blend Space** - Displays information relating to MxM's blend space system

12.2 Recording and Playback

Without recording, the debugger will not display any data. To begin recording, click on the red dot 'record' icon in the timeline toolback. This will dynamically record all data from every frame into a circular buffer. To stop recording, click on any of the timeline toolbar buttons.

By clicking and dragging anywhere in the graph view, the position of the playhead frame can be changed. Data displayed in the data view will show the recorded data at the playhead frame.

To preview the recorded animation data in the game, press the 'Preview' button on the timeline toolbar. The MxMAnimator will be set to the recorded state specified by the playhead frame. While 'preview' mode is on, moving the playhead or playing the recording, will change the animation state in game.

12.2 Pose Masks

Pose masks are used to mask out poses in the animation database that are never used for performance. The only way to determine what poses are not being used is to continually play and record animation in the debugging window. Note that the pose mask data is not limited to the size of the recording circular buffer.

After recording in the debugger for a significant period of gameplay time, a pose mask can be baked and applied to your animation data by clicking the 'Save Pose Mask' button in the top right corner of the debug window. This will automatically generate a pose mask file and attach it to your anim data. Note that this is not permanent and the pose mask can be removed. It is only recommended to create a pose mask later in development for optimisation purposes.

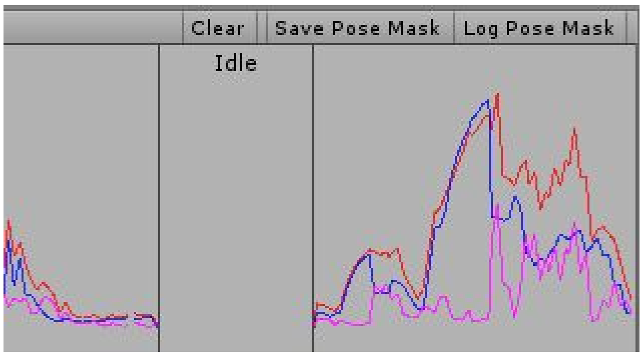
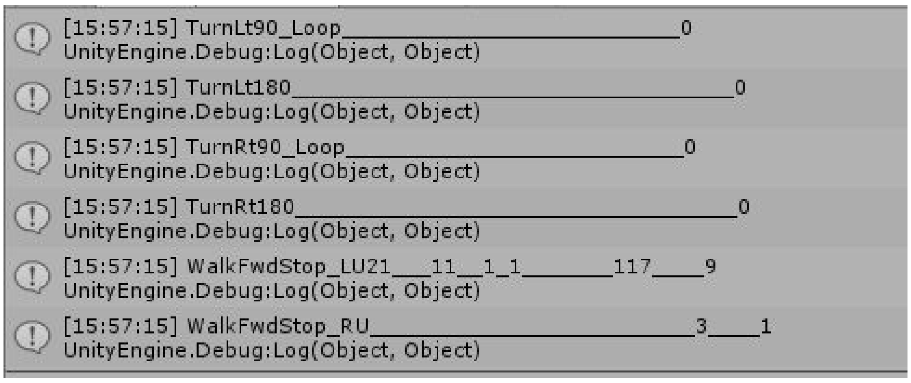


Figure 12.3 ‘Save Pose Mask’

In addition to saving pose masks, the data can simply be logged to the Unity console to see what animations are being used the most or least. To do this, click on the ‘Log Pose Mask’ option in the top right corner of the debug window.



```
[15:57:15] TurnLt90_Loop__________________________0
UnityEngine.Debug:Log(Object, Object)
[15:57:15] TurnLt180__________________________________0
UnityEngine.Debug:Log(Object, Object)
[15:57:15] TurnRt90_Loop__________________________0
UnityEngine.Debug:Log(Object, Object)
[15:57:15] TurnRt180__________________________________0
UnityEngine.Debug:Log(Object, Object)
[15:57:15] WalkFwdStop_LU21___11__1_1_______117____9
UnityEngine.Debug:Log(Object, Object)
[15:57:15] WalkFwdStop_RU_________________________3____1
UnityEngine.Debug:Log(Object, Object)
```

Figure 12.4 ‘Pose Mask logged to console’

Each debug line prints the name of the animation and a series of digits, each digit representing a pose in the animation. The last digit represents the number of poses actually used while intermediate digits represent their frequency of use on a scale of 1 - 9. Underscores ‘_’ represent un-used poses.

13 Optimisation

‘Motion Matching for Unity’ is already lightning fast thanks to an implementation with Unity’s job system, burst compiler and SIMD mathematics library. MxM also does a number of things under the hood to improve performance so you don’t have to think about it. There are, however, some considerations when it comes to optimisation worth noting.

13.1 Update Rate

It is not necessary to update the MxM pose search every single frame unless you are running on a low frame rate. In fact sometimes high update rates can lower quality. I recommend an update rate between 10Hz and 60Hz. Default setting is 30Hz which work well for most characters.

13.2 LOD & Frustum Culling

MxM currently doesn’t contain a built in LOD system (this is a planned future feature). However, it can still be implemented by dynamically modifying the update rate at runtime through function calls to the MxMAnimator component. Characters far away can be updated much less frequently than the player or other players close by. Also, if a character is not in view of the camera frustum it shouldn’t be updated at all.

Keep in mind that you cannot use LOD or frustum culling on characters that use root motion as it will change their movement behaviour. Note that I do not recommend using root motion for character movement.

13.3 Matching Metrics

The more you try to match the slower it will be. Only try to match what you need. For general locomotion you only need to match three joints and about 4 trajectory points. MxM supports more, but avoid matching too much if need additional optimisation. 3-4 joints, 4-5 trajectory points is a good rule of thumb unless you have a special case.

13.4 DoNotUse Tag

Poses that are tagged with DoNotUse will automatically be culled from the search resulting in increased performance. This is more of a side effect than an active optimisation measure, though it is good to be aware of it.

13.5 Pose Masks

A pose mask can be generated to cull a significant number of poses based on what is actually used by the system. This is recommended only later in development. Typically, for a 45 minutes set of mocap data, the system will only need / use 20% of it. The other 80% can be culled. This is done by training the system while playing the game over a period of time so it can be time consuming to generate.

13.6 Tag Separate Bins

MxM doesn’t search every single pose in your animation database, at least not if you are using require tags. Poses are automatically sorted into search bins based on their combinations of tags. Only the bin matching the current required tags will be searched. Therefore, using require tags to separate clearly different animation sets (e.g. combat - locomotion - crouching) can help improve performance.

13.6 Next Pose Tolerance Test

Sometimes it is not necessary to do a motion matching search. This is particularly true if the character is running straight and not changing animation. The ‘Next Pose Tolerance Test’ option on the MxMAnimator component can be turned on to run a test that checks if the next pose in the current animation is “Close Enough” to the desired trajectory outcome. If this test passes, the pose search is abandoned completely and the current animation keeps playing.

This saves a significant amount of performance particularly when there are multiple motion matched characters in the scene.

13 Troubleshooting

This is a live document and as such it will be updated as the MxM is expanded and as feedback is provided. This section will contain troubleshooting advice for common issues.

Problem: Sometimes the character glitches out into the ‘muscle space’ pose as if it has no animation.
Solution: Try reducing the update rate of MxM. If you are running on high FPS, updating every frame can be problematic as too many animations try to blend in. Keep update rate between 0.033s (30Hz) and 0.2s (5Hz).

Problem: My character moves slow and jittery when using the MxMTrajectoryGenerator.
Solution: Try the following:

1. Ensure that ‘root motion’ is set to ‘Handled by Motion Match Entity’ in the ‘Options’ tab of the ‘MxMAnimator’ component.
2. Ensure that there is a standard unity ‘Character Controller’ component attached.
3. Remove any rigid bodies or other capsule colliders on your character.

Problem: My character keeps playing the start of a moving animation but never gets into a running loop. It accelerates and decelerates constantly(MxMTrajectoryGenerator)
Solution: Try the following:

1. Ensure that the max speed of your MxMTrajectoryGenerator is high enough to match the speed of your fully running animation. Even slightly faster so the green trajectory goes a little bit past your run animations red trajectory.
2. Ensure that you have added the correct animations into the pre-processor and have pre-processed it.
3. Ensure your run loop animation fits seamlessly with your acceleration animations (continuity)
4. Check your calibration. You may need more weight on the trajectory. Try moving the Pose-trajectory slider to the right. Also try reduce ‘Velocity Costing between 0.15 - 0.25.
5. Explicitly give the run loop animation favour using pose favour utility tags in the MxMTimeline. Set the favour value between 0.8 - 0.9 for the run loop poses.

Problem: My character stays in idle and won't move.
Solution: Ensure that the object used as the 'Target Model' in the 'General' foldout of your pre-processor is an actual model file (e.g. an fbx) with a valid rig. Then redo the preprocessing. Do not put a prefab in this slot.

Problem: My character feels very unresponsive.
Solution: Try modifying the calibration of your MxMAnimData to give the trajectory a higher weighting compared to the pose. Use the Pose-Trajectory slider for this. Additionally, it should be noted that motion matching can only be as responses as the animations you feed it. Make sure your animations are authored or modified to match the gameplay you want.

Problem: My characters motion doesn't feel very fluid.
Solution:

1. Try modifying the calibration of your MxMAnimData to give the pose a higher weighting compared to the trajectory. Use the Pose-Trajectory slider for this.
2. Note that motion matching can only be as fluid as the coverage of your animations. If you are missing transitions and coverage then MxM will have to jump large gaps in poses which can reduce fluidity.
3. Try using the blendspace system to cover gaps in coverage where possible (section 5)
