



synty®

Base Locomotion

Keyframe Animation pack

User Guide

Version: 1.0.0

www.syntystore.com

Table of Contents

1. Introduction	3
2. Installation and set up	4
Requirements	4
3. Animation Pack Components	4
Key Features	5
List of Animations	6
Understanding Animation Naming Conventions	7
Tips for working with Animations	7
Optimizing Animation Performance	7
4. Quick Start	9
Demo Scenes	9
Demo_01	9
Gallery_01	10
Build your own	11
Applying Animations to characters	11
Swapping Animation sets between Masculine and Feminine	11
Creating a new Avatar	12
Modifying existing Avatar	12
Control Mappings	13
Naming conventions	13
5. Sample Animation Controller	14
Animation Transition Logic	16
Blend Trees	16
Animation Layers	17
Animation Parameters and Triggers	17
Examples for Mapping Animations to States	18
States	18
Locomotion	18
Crouching	18
Slope Detection	19
Strafing	19
Lock-On	20
Starts/Shuffles	20
In Air	21
Turn-In-Place	21
Additive Layers	22
Setup and Integration	23
Camera Integration	24
6. Character Avatar	25
Mecanim Humanoid Character Avatar	25
Adjusting Avatar Properties	26
7. Acknowledgements of copyrights	27
8. Terms of use	27
9. Glossary	29

1. Introduction

Welcome to Base Locomotion, a specialised animation pack engineered for seamless integration across various platforms, ensuring swift implementation into your preferred development environment. The pack comes with animations curated to accelerate your animation workflow, providing a collection of seamlessly integrated motions that elevate the performance of humanoid characters to a new standard. Witness your characters come to life with a fluidity that not only enhances realism but also serves as a catalyst for boundless creative exploration and precise character navigation.

From all of us at Synty,
We thank you for the support!

2. Installation and set up

Requirements

Animation component files

- Unity supports fbx imports across all current LTS versions

Sample content

- Unity Version 2021 LTS+ or later
- Input System Package 1.5.1 or later
- Basic Locomotion is a collection of fbx animation files that have a wider compatibility across engines and animation tools, however to function correctly with the Animation Controller, Character Controller and Input System please ensure you follow the system requirements and the instructions in the relevant sections below that cover those topics.

3. Animation Pack Components

The Animation Pack features 205 purpose built animations, an ideal base for building a seamless humanoid animation system. With comprehensive coverage, it includes movements such as crouching, running, walking, sprinting, and standing, ensuring fluid transitions between states.

Key features

- Animation Sets:
 - 100 Masculine Animations and 100 Feminine Animations, designed for smooth blending, allowing for uninterrupted transitions between various states. Additionally, there are 5 Neutral additive animations that work with both sets.
- Humanoid Character Avatar:
 - These animations seamlessly integrate with Unity's Mecanim system, providing a foundation for easy to integrate character animations across different characters.

- Sample Animation Controller:
 - Animation Controller's state-based system to organize and sequence character actions. Achieve a structured and modular approach to animation sequencing, allowing for seamless transitions between states and ensuring a fluid and responsive character experience.
- Sample Animation Controller Script:
 - Organize character parameters in a single script, enabling a structured and clear approach to tuning character movement. Transition seamlessly between states for fluid character movements.
- Demo Scene:
 - Scene built with various parts of the character connected as a sample of how a user could potentially use the content for a playable character
- Gallery Scene:
 - Scene of all the animations as separate assets for users to view clearly as individual animations.

List of Animations

Idle A_Idle_Crouching A_Idle_Standing InAir A_InAir_FallLarge A_InAir_FallShort A_Jump_Idle A_Jump_Running A_Jump_Sprinting A_Jump_Walking A_Land_ToldleHard A_Land_ToldleMedium A_Land_ToldleSoft A_Land_Idle A_Land_Walking A_Land_Running A_Land_Sprinting Locomotion A_Crouch_BckStrafeB A_Crouch_BckStrafeBL A_Crouch_BckStrafeBR A_Crouch_BckStrafeFL A_Crouch_BckStrafeL A_Crouch_FwdStrafeBR A_Crouch_FwdStrafeF A_Crouch_FwdStrafeFL A_Crouch_FwdStrafeFR A_Crouch_FwdStrafeR A_Walk_BckStrafeB A_Walk_BckStrafeBL A_Walk_BckStrafeBR A_Walk_BckStrafeFL A_Walk_BckStrafeL A_Walk_BckStrafeR A_Walk_FwdStrafeBR A_Walk_FwdStrafeF A_Walk_FwdStrafeFL	A_Walk_FwdStrafeFR A_Walk_FwdStrafeL A_Walk_FwdStrafeR A_Walk_F A_Walk_Down25F A_Walk_Up25F A_Run_BckStrafeB A_Run_BckStrafeBL A_Run_BckStrafeBR A_Run_BckStrafeFL A_Run_BckStrafeL A_Run_BckStrafeR A_Run_FwdStrafeBR A_Run_FwdStrafeF A_Run_FwdStrafeFL A_Run_FwdStrafeFR _Run_FwdStrafeL _Run_FwdStrafeR A_Run_F A_Run_Down25F A_Run_Up25F A_Sprint_F A_Sprint_Up25F A_Sprint_Down25F A_Shuffle_Standing_F A_Shuffle_Standing_L A_Shuffle_Standing_R A_Shuffle_Standing_B A_Shuffle_Crouching_F A_Shuffle_Crouching_L A_Shuffle_Crouching_R A_Shuffle_Crouching_B A_Turn_Standing_90L A_Turn_Standing_90R A_Turn_Standing_180L A_Turn_Standing_180R	A_Turn_Crouching_90L A_Turn_Crouching_90R A_Turn_Crouching_180L A_Turn_Crouching_180R Transitions A_Crouch_ToStand A_Stand_ToCrouch A_Sprint_ToCrouch A_Sprint_ToCrouchRootMotion A_Idle_ToRun90L A_Idle_ToRun90LRootMotion A_Idle_ToRun90R A_Idle_ToRun90RRootMotion A_Idle_ToRun180L A_Idle_ToRun180LRootMotion A_Idle_ToRun180R A_Idle_ToRun180RRootMotion A_Idle_ToRunF A_Idle_ToRunFRootMotion A_Run_ToldleF_LFoot A_Run_ToldleF_LFootRootMotion A_Run_ToldleF_RFoot A_Run_ToldleF_RFootRootMotion A_Idle_ToWalk90L A_Idle_ToWalk90LRootMotion A_Idle_ToWalk90R A_Idle_ToWalk90RRootMotion A_Idle_ToWalk180L A_Idle_ToWalk180LRootMotion A_Idle_ToWalk180R A_Idle_ToWalk180RRootMotion A_Idle_ToWalkF A_Idle_ToWalkFRootMotion A_Walk_ToldleF_LFoot A_Walk_ToldleF_LFootRootMotion A_Walk_ToldleF_RFoot A_Walk_ToldleF_RFootRootMotion	Neutral * A_Lean_Additive_Neut A_BodyLook_Additive_Neut A_HeadLook_Additive_Neut A_TPose_Neut
--	---	--	--

The animations included in this package are organized into two distinct sets: one featuring masculine actions and the other feminine. Both sets share an identical list of animations, maintaining a consistent structure throughout. Importantly, each animation within its respective set seamlessly blends with others, ensuring a smooth and cohesive transition between different actions. This deliberate organization provides users with flexibility and coherence when integrating animations into their projects.

*** The Neutral animations work with both sets as additive animations for body leaning, head and body looks.**

Understanding Animation Naming Conventions

Example: A_Run_BckStrafeBL_Masc

<Filetype> _ <AnimationType> _ <Description> _ <AnimationSet>

- Filetype: The broad-scope prefix. Here 'A' stands for 'Animation'
- AnimationType: The general category of animation. Here this is a 'Run' animation
- Description: 'BckStrafe' specifies that this animation uses a 'backwards strafe' running cycle, and the last 'BL' indicates that it is in the specific direction 'Back-Left' (i.e. at a diagonal).
- AnimationSet:
 - 'Masc' stands for 'Masculine',
 - 'Femn' stands for 'Feminine'
 - 'Neut' stands for 'Neutral'

Tips for working with Animations

Optimizing Animation Performance

Unity has numerous options to help optimize performance when using animations. These options are within the animation tab for each fbx animation asset.

Import Settings:

- Use the "Rig" tab to configure Avatar Definition and Animation Type settings.
- Enable "Optimize Game Objects" to remove redundant Transform components.
- Consider enabling "Import Blend Shapes" if needed.

Animation Compression:

- Utilize Unity's animation compression options.
- Use "Optimal" for keyframe reduction without sacrificing visual quality.
- Experiment with different compression ratios to find the right balance between performance and quality.

Remove Scale Curves:

- If not needed, consider removing scale curves from animations.
- Scale curves can impact performance, and removing them can streamline the animation data.

4. Quick Start

Demo Scenes

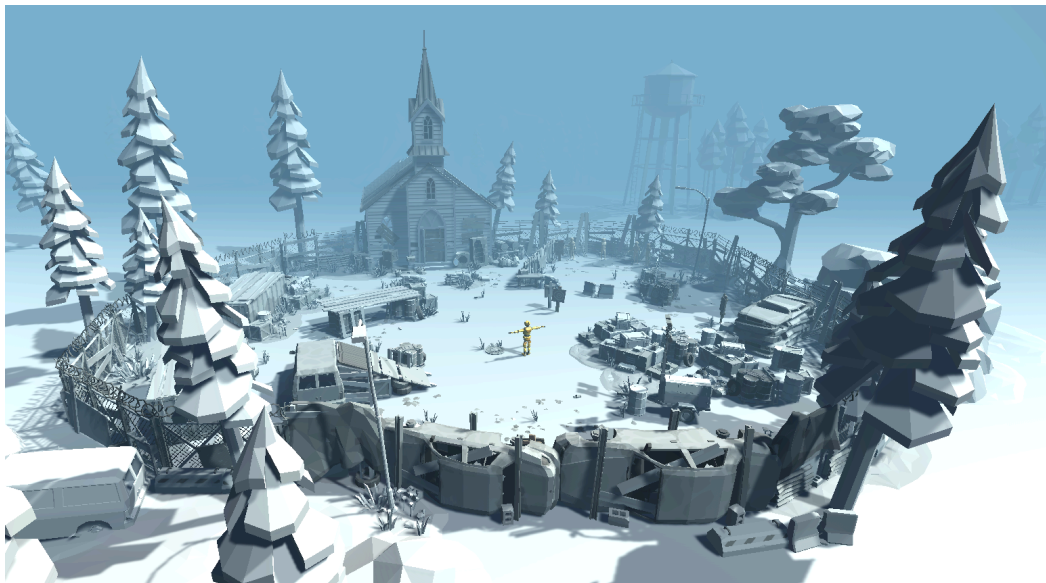
We have provided a few Demo scenes for you to see how you might use Synty Dynamics. These below sections describe each demo scene and what they demonstrate.

You will find these scenes in **Assets/Synty/AnimationBaseLocomotion/Scenes**

Demo_01

This scene demonstrates the use of the various systems in combination to create an animated moving character. The scene demonstrates:

- mecanim compatible Avatar using the Character Controller Component
- Modular State Machine driven Animation Controller
- Versatile Input options for control via Mouse/Keyboard or Gamepad
- Movement Toggle to choose between 'Always Strafing' or 'Input Direction'
- Simple lock-on system for targeted interactions



As seen in the image is the compound space for a user to run around as the character dummy to test various movements like walking, running, sprinting, strafing, jumping, crouching/sliding under objects and locking on to targets.

This scene can be used to test animations or other characters on the Synty Locomotion system as a quick start to getting a character moving.

Gallery_01

This scene demonstrates the various animations in this pack as a virtual gallery/library so users can break down how the animations work in isolation, to understand further how they work in tandem:

- Labelled groups of animations to quickly track down specific animations
- Looped to see timing/length of animation



The virtual gallery constitutes a controlled environment designed for the presentation and examination of a collection of animations shaping the character animator's work. Within this space, users can assess the full spectrum of animations in a clear and uncomplicated manner.

Build your own

The following example will walk you through a typical use case of **Base Locomotion** as a means to guide your experimentation further to find your own use cases. This assumes you have installed it correctly and starting from a new scene.

Applying Animations to Characters

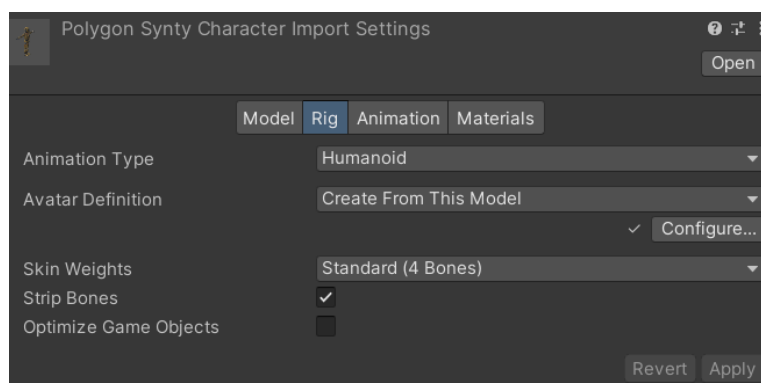
In Unity, a developer can import the animation package into a new scene and reference the **PolygonSyntyCharacterAvatar** to retarget the animation to their own character. This character can be a Synty character or a completely different biped, however the animations will work the best with Synty characters as the animations were created with their proportions in mind.

Import Characters:

- Import Base Locomotion package into project for access to the animations and **PolygonSyntyCharacterAvatar**
- Import new character that will be the target for Base Locomotion animations

Create a Humanoid Avatar on new character:

- On the new character, create a new Avatar and configure bone mappings



Test that the mappings are correct (optional)

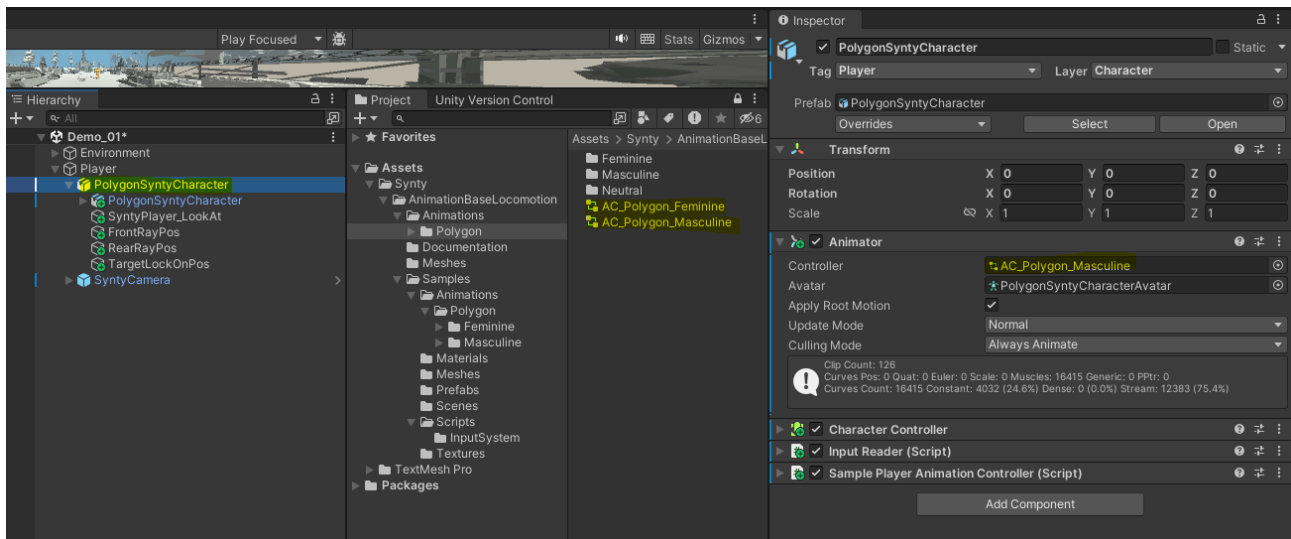
- In the **Demo_01** scene within the Base Locomotion pack, drag the new character into the hierarchy of the **Project** tab, under

Player/PolygonSyntyCharacter and delete the

Player/PolygonSyntyCharacter/PolygonSyntyCharacter asset

- In the Inspector for the **Player/PolygonSyntyCharacter** in the **Project** tab, apply the Avatar of the new character to the Avatar in the Animator Component

Swapping Animation sets between Masculine and Feminine



- Navigate to the Inspector window on the **PolygonSyntyCharacter**
- Drag in **AC_Polygon_Feminine** or **AC_Polygon_Masculine** into the Controller on **PolygonSyntyCharacter**

Creating a new Avatar

Animation Type:

- Navigate to the Inspector window in Unity of the Character.
- Click on the **Rig** tab to access the Avatar Configuration tab.

Avatar Definition:

- Set the Animation Type to **Humanoid**
- Set the Avatar Definition to **Create From This Model** to generate a humanoid avatar based on the character's rig.

Modifying existing Avatar

Access Avatar Configuration:

- Click on **Configure...** to access the Avatar Configuration tab.

Bone Mapping:

- Review and adjust bone mappings to ensure precise alignment with the character's skeletal structure.

Preview and Apply:

- Check for any errors that occur or if you are using the same bone in two definitions in the skeletal mapping
- Apply changes to update the avatar properties.

Control Mapping

Although fully customisable using the **Controls** asset inside the **Synty\AnimationBaseLocomotion\Samples\Scripts\InputSystem** directory, the demo mapping is listed below.

Keyboard:

- Move: WASD or ↑ ← ↓ →
- Look: Mouse Input
- Jump: Space
- Crouch: Ctrl
- Sprint: Shift
- Toggle Walk: CAPS
- Toggle Lock-On: Middle Mouse Button

Gamepad:

- Move: Left Stick
- Look: Right Stick
- Jump: Button South
- Crouch: Button East
- Sprint: Right Shoulder
- Toggle Walk: Left Stick Press
- Toggle Lock-On: Right Stick Press

Naming conventions

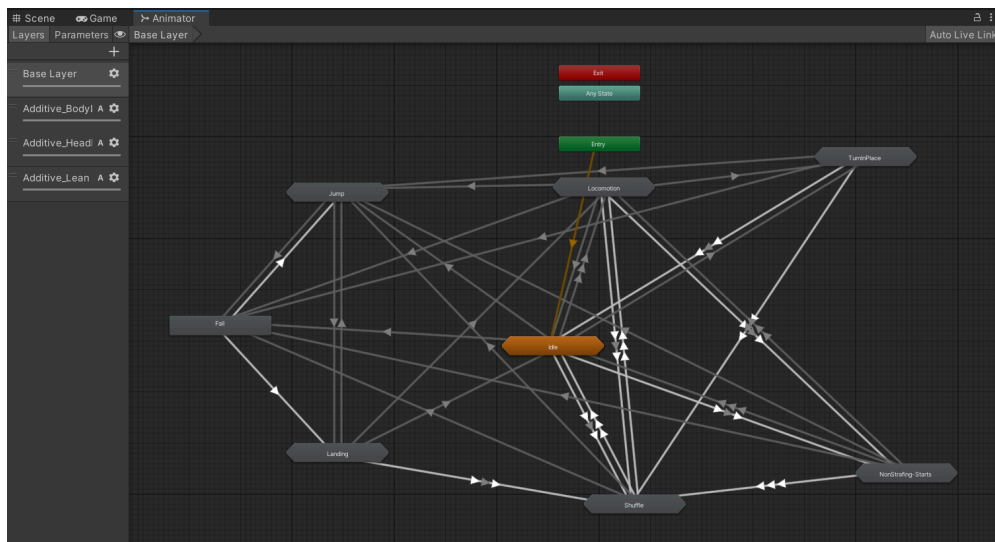
A_	Animation
AC_	Animation Controller
M_	Material
PM_	Physics Material
SK_	Skeletal Mesh
SM_	Static Mesh
T_	Texture

Scripts are contained in the `Synty.ProductName` namespace.

5. Sample Animation Controller

The Animation Controller serves as the orchestrator of dynamic character movements within Unity. This sample component within the Animation Pack offers an example of the framework for controlling and sequencing animations. Further reading for reference can be found in the Unity User Manual ([Manual: Animator Controller](#))

Otherwise here is a concise overview of the Animation Controller in context with the sample provided:



State-Based System:

- Organizes animations into states, allowing for seamless transitions between different character actions, from idle stances to complex movement sequences.

Transitions:

- Manages transitions between states, providing a smooth and clear approach to changes in the animation being played. Transitions can be triggered by conditions such as user input, script events, or animation completion.

Blend Trees:

- Introduce flexibility through Blend Trees, enabling smooth interpolation between animations based on parameters like speed, direction, or any custom variable.

Layers:

- Supports layering, allowing the overlay of animations for added complexity. This is particularly useful for simultaneous actions like upper body movement during locomotion.

Parameters:

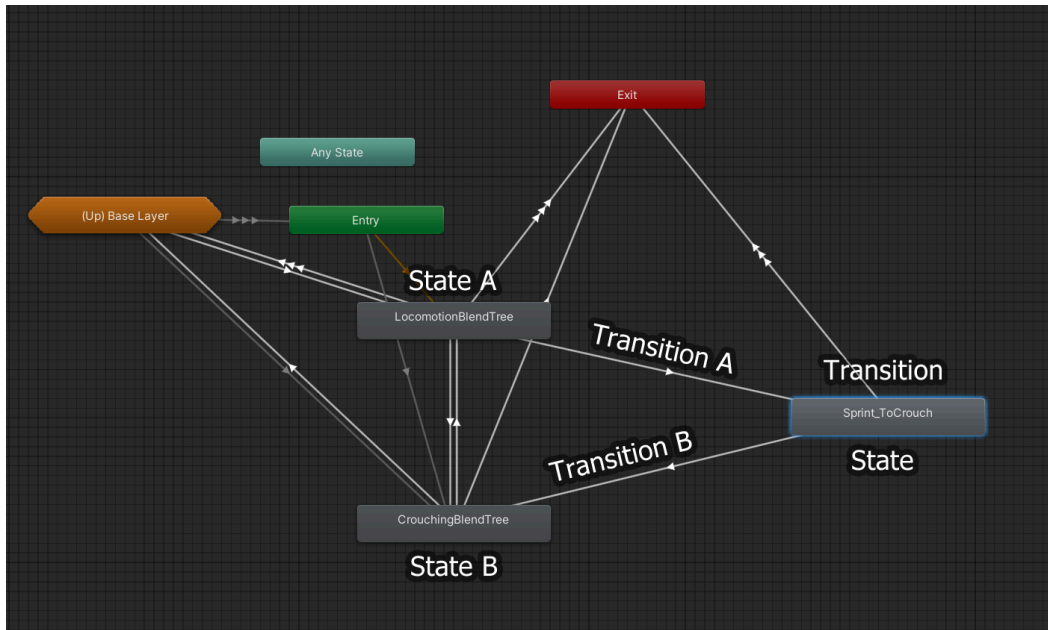
- Utilizes parameters and triggers to dynamically influence animation states, offering a versatile means to adapt character animations to varying in-game conditions.

Animation Transition Logic

In Unity's Animation Controller, transitions between animation states are managed by right-clicking a state and choosing 'Make Transition'. These transitions are governed by specified conditions set in the Inspector window, using parameters (booleans, floats, integers etc) from the Animation Controller. ([Manual: Animation transitions](#))

State A - (*TransitionA*) > Transition State - (*TransitionB*) > State B

Example: Adding a slide in between sprinting and crouching.



Transition from Locomotion to Sprint_ToCrouch, and another from Sprint_ToCrouch to Crouch

- *TransitionA* Conditions:
- **CurrentGait** Equals 3 (Integer)
- **IsCrouching** true (boolean)
- **Has Exit Time** (unchecked)
- *TransitionB* Conditions:
- **IsGrounded** true (boolean)
- **IsJumping** false (boolean)
- **Has Exit Time** (checked)

In *TransitionA*, both conditions must be met (**CurrentGait** is 3 and **IsCrouching** is true). **Has Exit Time** is unchecked, allowing interruption of the Locomotion state animation (*StateA*). *TransitionB*, with **Has Exit Time** checked, transitions to the Crouch state (*StateB*) only when specific conditions are met after the Sprint_ToCrouch animation completes.

Blend Trees

Blend trees are a tool within the Animation Controller that supports smooth transitioning between different animations based on the values of specified

parameters. It's often used to seamlessly blend animations, like transitioning from walking to running based on the character's speed.

Unity has a variety of blend types within a blend tree that a user can modify or switch to depending on the purpose of the animation they wish to implement.

([Manual: Blend Trees](#))

Animation Layers

Animation layers are a robust tool that allows developers to control and blend multiple animations on a GameObject independently. These layers operate on a priority system, ensuring smooth transitions between different sets of animations with higher priority layers taking precedence. This helps with structuring animations by isolating specific groups of animations that can then be driven dynamically through the parameters across each layer universally or individually. Accessing the Layers Widget allows a user to set and define a mask for that layer, and the blending type for how the animations in that layer are applied. 'Override' replaces animations on previous layers with those on the current layer, 'Additive' will superimpose the animations on top with the previous layers to dynamically calculate the new animation to be played.

([Manual: Animation Layers](#))

Animation Parameters and Triggers

Animation parameters are variables that are defined within an animation controller. These dynamic values can access and change the state machines behavior. These parameters can be updated by animation curves and/or accessed from scripts, allowing control over various aspects, animation, blending, or even other peripherals like sound effects or interactions. Scripted modifications to parameters are necessary for interactions with Mecanim, where a parameter set by a script can control a blend tree, providing precise control of the animation output. Parameters can be

- Integer (whole numbers)

- Float(numbers with fractions)
- Bool (true/false)
- Trigger(a bool reset by the controller)

The values for parameters are set in the Animator window's Parameters section, allowing animators to customize the default settings.

([Manual: Animation Parameters](#))

Examples for Mapping Animations to States

The following information is with the **SamplePlayerAnimationController** in mind as an example of how to create and implement animations and states within Unity, but these are merely suggestions of ways someone could implement content from this pack, it is not the only way to do so.

States

The **SamplePlayerAnimationController** script is responsible for managing the Player's states and values. The current active state is stored in the **CurrentState** variable, and using a State's **SwitchState** method, different behaviours can be implemented.

For example, the character is in the **LocomotionState** most of the time, but whenever it is no longer grounded, it enters the **FallState**, which applies an additional **ApplyGravity** method via **Update()**. Similarly, **JumpingState** allows for custom behaviour during the ascending part of a jump before it becomes a **FallState**, and **CrouchingState** allows for control over behaviour during crouching.

This setup gives you the flexibility to create new states and behaviours to suit the needs of your project.

All variables which need to persist consistently across state changes live within the **SamplePlayerAnimationController** script. These can be tweaked in the inspector as the script is a component of the **SyntyCharacter** game object.

Locomotion

There are 3 different locomotion speeds or 'gaits' with their own animation sets:

- Walk (1.4 m/s)
- Run (2.5 m/s)
- Sprint (6 m/s)

The default speeds can be modified on the **SamplePlayerAnimationController** in the Inspector when it is attached to a game object.

Crouching

Crouching uses the same locomotion speed as 'Walk'. There is a custom slide animation that is set up to trigger from a 'Sprint' speed when the 'Crouch' key is pressed as a transition animation, ending in the Crouch pose.

Crouching also reduces the capsule height so the player can pass underneath lower objects. There is a 'CeilingHeightCheck' method in 'PlayerBaseState' which is used to check if the player is able to stand up from a crouch. If not, the crouching state is maintained.

Slope Detection

The controller has basic slope-detection which allows an 'uphill' or 'downhill' version of locomotion to be played inside a blend tree in the animator, controlled by the parameter 'InclineAngle'. This angle is calculated using two empty game object positions under the player hierarchy: FrontRayPos and RearRayPos. These two points (which can be moved around and customised) are used as the origin positions for ray casts by the 'CheckInclineAngle' method in the 'PlayerBaseState' script. The angle between where the two rays collide with the ground gives the angle of the ground, relative to the player's forward vector.

Strafing

The 'Sprint' gait does not include strafing. However, Walks, Crouch, and Run locomotion all have two 8-directional blend trees setup for strafing:

- **StrafeBackwards_BlendTree**
- **StrafeForwards_BlendTree**

In the animator, the parameter **ForwardStrafe** is a float that blends between 0 and 1 to smooth the transition between blend trees.

This setup serves to minimise any problems when the player input could sit at a strafing direction that is halfway between two animations which have opposing (Forward vs Backward) animation cycle directions (for example, between 'Strafe R' and 'Strafe BR') . Typically a 50% blend between two such animations can result in legs intersecting each other.

For this reason, this pack includes (for example) two of each 'Strafe R' and 'Strafe BR' animations, one with forwards locomotion and one with backwards locomotion.

Any player locomotion taking place at an angle between R and BR will thus look correct regardless of whether it is using the Forward or Backward blend trees in a given moment. This window allows time for the BaseState logic to check the strafing angle, and when a threshold is passed it can switch to the Backward or Forward blend tree.

Additionally, there are two L and FL strafe animations (Forward and Backward) to create an overlap window on the other left directional side of the blend trees.

Lock-On

The demo controller includes a simple 'lock-on' feature which will focus the player on an object in the world and keep it centred, allowing the player to strafe around and always be aiming at the object.

The demo also includes an example target object **PF_LockOnTarget** This has an **SampleObjectLockOn** script that can be used to make any game object a viable lock-on option to the player.

When the player (with Tag 'Player') enters the object's collider area, the **SampleObjectLockOn** script adds the parent object to the player's list of viable targets. Then the player state machine has logic that determines the best viable target from all those within range, and allows locking-on to be toggled.

Starts/Shuffles

The animator uses the variable **InputTapped** to trigger a start animation from idle. If the player is strafing, the controller uses the 'Shuffle' animations (a blend tree between the directions: L R F B). If the player is not strafing, the controller

plays a 'Start' animation ranging from a 180 degree turn, to a directly forward start.

Note: **InputHeld** is a variable used to indicate that the input/key is still held and the start animation should transition to locomotion.

In Air

You can control the feel of the jump by adjusting the 'JumpForce' value in the Inspector on the **SamplePlayerAnimationController** script (a higher value results in a larger jump). You can also control the **GravityMultiplier** value to make the falling faster or slower.

This pack includes custom jump animations for each gait (idle, walk, run, sprint) animated fully from start to end (i.e. a run, flowing into a jump, a land, and back to a run). These are then broken into separate clips, so each has a specific 'jump' and a 'land' clip to be utilised by the animator.

Depending on what speed the player is travelling, the 'CurrentGait' integer variable on the **SamplePlayerAnimationController** keeps track of which gait the player is closest to. In the animator, this variable is then used within transition rules to select which 'jump' and which 'land' animations to play, so a running character will always play the running jump and landing, for example.

There is also a 'FallingDuration' variable which is used to determine whether a 'soft', 'medium' or 'hard' landing animation should play. The more time spent in-air, the harder the landing.

The 'FallingDuration' also controls a blend tree for the 'InAir' animation, to gradually transition from a 'Short' distance falling animation loop to a 'Large' distance falling animation loop.

Turn-In-Place

The Demo controller has a 'turn-in-place' blend tree which uses the root motion of the 90 degree turn animations to rotate the player towards the camera's

facing direction while strafing is enabled.

The parameter **CameraRotationOffset** measures the difference in angle between the player and the camera, and the blend tree uses this value to determine what percentage of a full 90 degree turn should be played.

Additive Layers

The animator uses additive animation layers to enhance the fluidity of the character's locomotion. These layers are:

- **Lean**
- **HeadLook**
- **BodyLook**

The lean additive will offset the player animation so it appears to lean inwards towards the direction that the player is turning, and the body look and head look offsets the rotation of these body parts so it appears the player is anticipating/looking ahead in the direction of the turn.

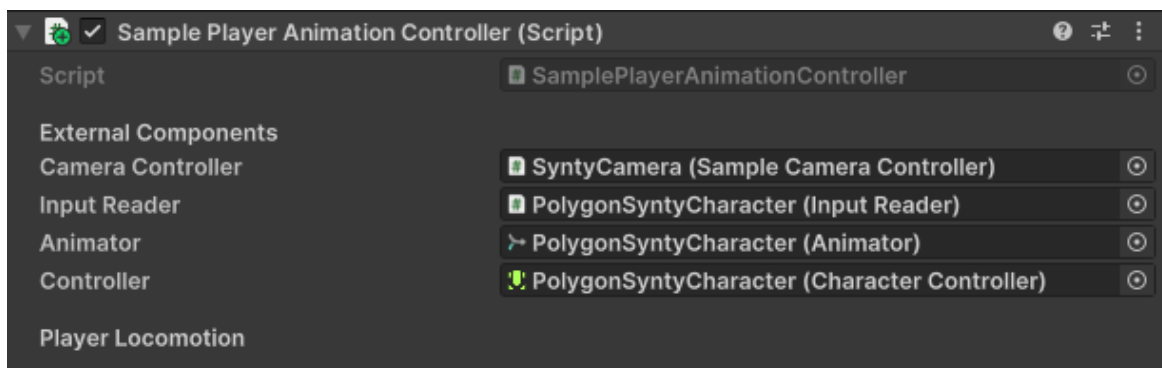
These additive layer weights are driven by the rate at which the player is turning. The faster the turn rate, the more the layer is applied. This relationship is also controlled by a curve in the inspector so the falloff can be fine-tuned. These curves are adjustable on the PlayerStateMachine in the inspector:

- **LeanCurve**
- **HeadLookXCurve**
- **BodyLookXCurve**

The up and down look (Y axis) is driven by the camera angle - if the camera is tilted to look up or down, the additive layers apply a little rotation to make the player look up or down accordingly.

Turns and Leans can be enabled or disabled within a state (e.g. PlayerCrouchingState) with the 'CalculateRotationalAdditives' method, which takes boolean inputs to potentially override Lean, HeadTurn, BodyTurn in that order. For example, in 'PlayerCrouchingState' the method has lean and body turn set to 'false' and thus disabled.

Setup and Integration



This sample is set up to show some potential ways of connecting the various pieces to create an animated character. This is the Unity facing output of the script itself, which is written in C#. Ensure that the script is attached to your game object, and that it is properly configured to handle default values for movement variables for whatever purposes your character movement needs.

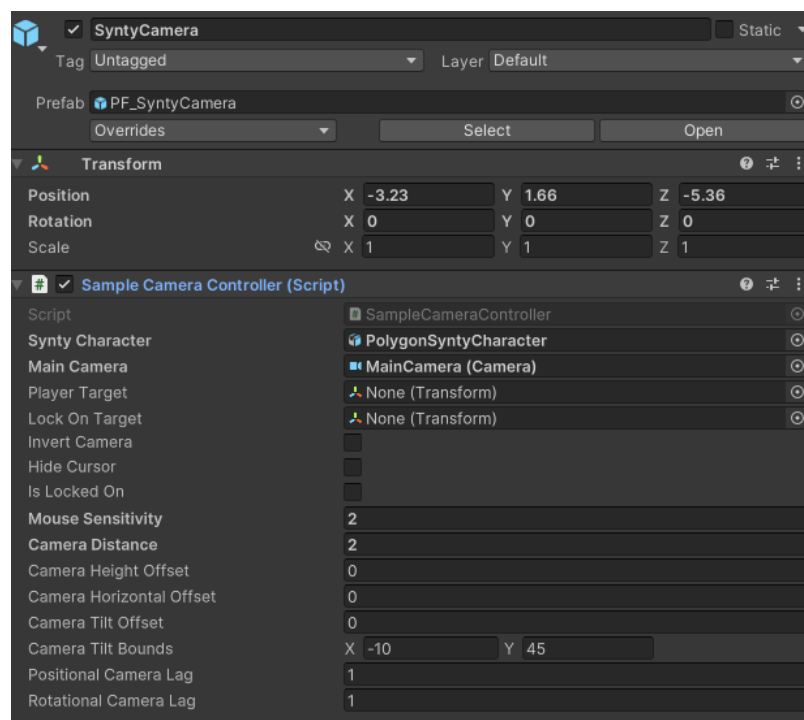
Next, establish connections with the following components:

- **Camera Controller:** Connect the player animation controller script to the **Camera** game object, allowing it to adapt character animations based on camera perspective or movement.
- **Input Reader** Link the player animation controller script to the input reader, enabling the script to interpret and respond to player input effectively.
- **Animator** Integrate the script with the Animator component. This involves linking the script to the Animator to control animations seamlessly.
- **Controller** Ensure that the player animation controller script is linked to the **PolygonSyntyCharacter**, enabling synchronization between animations and character movement.

Any further default values can be added within the script and then set within Unity as well.

Camera Integration

The **SyntyCamera** asset in the **demo_01** scene is set up to work seamlessly with the demo player controller. The **SampleCameraController** script has some simple functionality to alter the feel of the camera, and to handle lock-on targets in tandem with the **SamplePlayerAnimationController** script on the player.

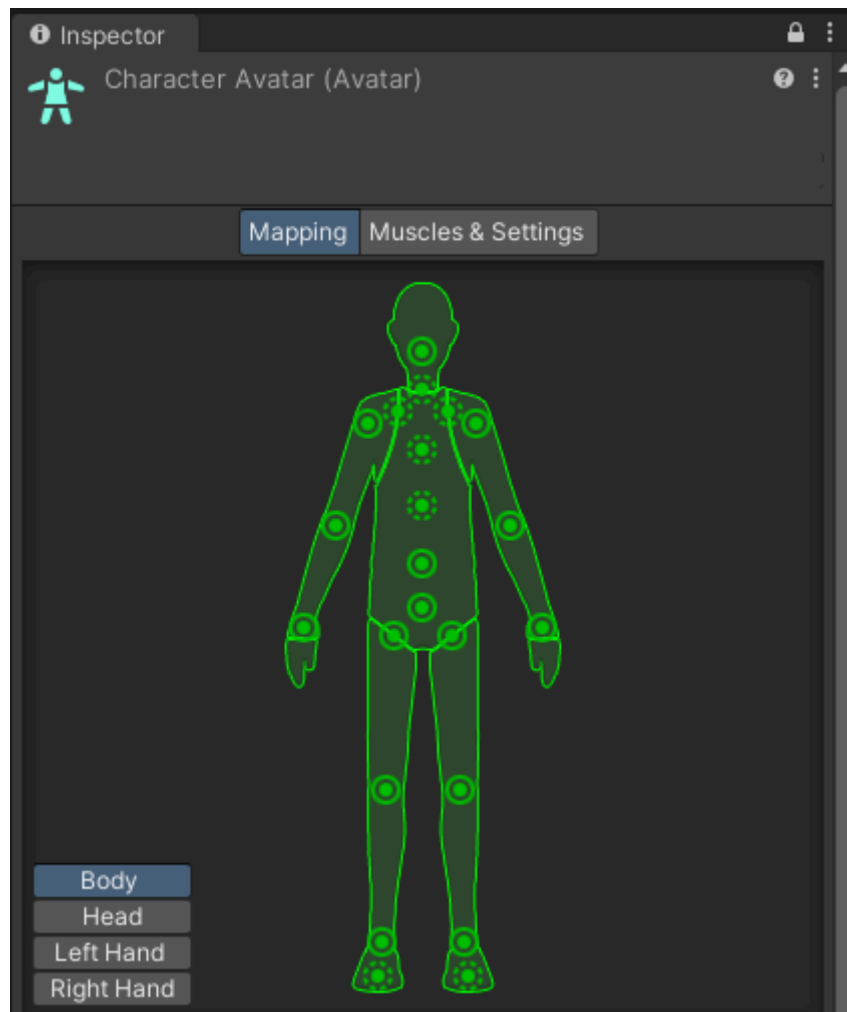


- **Player Target** The game object that the camera will follow – i.e. the player
- **Lock-On Target** A game object that when ‘lock-on’ is activated, the camera will aim towards
- **Invert Camera** Invert which input direction rotates the camera tilt up or down
- **Mouse Sensitivity** Adjust how rapidly rotation occurs mashed on mouse movement
- **Camera Distance** How far from the player the camera is positioned
- **Camera Height Offset** Offset the camera up and down from the player
- **Camera Horizontal Offset** Shift the camera to the left or right of the player

- **Camera Tilt Offset** Rotate the default angle of the camera up or down
- **Camera Tilt Bounds** Clamp the camera's tilt angle rotations to stop it rotating under the ground or flipping over the top of the player
- **Positional Camera Lag** Increase or decrease the positional delay with which the camera moves to follow the player
- **Rotational Camera Lag** Increase or decrease the smoothness with which the camera responds to its new target position

6. Character Avatar

Mecanim Humanoid Character Avatar



In Unity, the mecanim Humanoid Character Avatar serves as a fundamental framework for bipedal character animation. Essentially, an avatar in this context is a digital representation of a character's skeletal structure and body proportions. To access the avatar of a character you can find the mesh that the avatar definition is created from, in the case of the Synty pack, the Avatar is nested in the **Synty\AnimationBaseLocomotion\Meshes** as the **PolygonSyntyCharacterAvatar**.

Here's a breakdown to clarify its purpose for users:

- **Humanoid Structure:**
 - Unity's mecanim Humanoid Character Avatar adheres to a bipedal structure, aligning with the standard anatomy of human characters.
- **Compatibility Across Characters:**
 - Designed to be universally compatible, the avatar allows users to apply animations seamlessly to a variety of humanoid characters, streamlining the animation process.
 - Other character models on different bipedal rigs can be added to the project, with their own Avatar set up to allow the animations from this pack and others to be applied to them, bypassing the issue of compatibility.
- **Configurable and Adaptable:**
 - Users can configure their own avatars by adjusting the bone mappings and aligning them with Unity's Humanoid Avatar configuration.

Adjusting Avatar Properties

In Unity, within the rig tab of an asset you are able to define the skeleton type to one of the following, **Humanoid**, **Generic** or **Legacy**.

Humanoid is used when working with humanoid characters. It provides a standardized bone structure that makes it easier to work with humanoid animations, retargeting, and blending. The Humanoid rig is particularly useful when using Unity's Mecanim animation system.

Generic is a more flexible option that doesn't adhere to the humanoid bone

structure. It allows for more custom setups but may not be as compatible with certain features like retargeting humanoid animations.

Legacy is used for the older animation system in Unity. It's not recommended for new projects, as Unity has shifted its focus to Mecanim and the Humanoid rig.

The content within this pack is set up to be used with the Humanoid skeleton type, to leverage the aforementioned retargeting and blending with the Mecanim animation system.

7. Acknowledgement of copyrights

If we're using any trademarked/copyrighted materials (that we're allowed to use), acknowledge we don't own them with the appropriate owner referenced.

For example:

Xbox™ and the Xbox™ logos are trademark of **Microsoft® Corporation**

Nintendo Switch™, Joy-Con™ and their logos are trademarks of **Nintendo of America Inc.**

PlayStation™ is a registered trademark of **Sony Group Corporation**

8. Terms of use

The full terms of the End User License Agreement (EULA) apply and can be found at <https://syntystore.com/pages/end-user-licence-agreement>.

This is a summary of the license for the Synty [Product Name] software provided by Synty Studios Limited. This summary is for convenience only and is not legally binding.

Key Points

1. **License Grant:** When you purchase an Asset, you are granted a license to use the Asset subject to the terms of the EULA. All intellectual property rights in the Asset remain with Synty Studios Limited.
2. **Use of Asset:** You are entitled to incorporate the Asset into Products under your direct control, and into promotional materials for those Products. You can adapt the Asset for these purposes.
3. **Restrictions:** There are important restrictions on your use of the Asset. For example, you cannot use the Asset for Non-Fungible Tokens (NFTs), in Blockchain projects, Metaverse-related content, or with Generative AI Programs. You also cannot share or redistribute the Asset outside your team.
4. **Team Size and Seats:** Your license includes a limited number of seats for your team. If your team grows, you must purchase additional licenses.
5. **Unity Asset Store Purchases:** If you have purchased the Asset through the Unity Asset Store, the terms of the Unity Asset Store EULA also apply. You must consult the Unity Asset Store EULA for the full terms applicable to your purchase from the Unity Asset Store.
6. **Support and Source Files:** Support is provided at our discretion. You must not share source files of any Assets outside your team.
7. **Termination:** Your license can be terminated if you breach the EULA and fail to remedy the breach after notice from us.
8. **Warranties and Liability:** The Assets are provided "as is" without warranties of any kind, and our liability is limited.

For all the terms and conditions of the license, please read the full EULA available at <https://syntystore.com/pages/end-user-licence-agreement>. By using this Asset, you agree to be bound by the terms of the EULA.

9. Glossary

Additive – In animation, additive blending is a technique where animations are combined or layered on top of each other to produce a final animation. This is often used to add specific motions or details without affecting the underlying animation.

Animation Controller – An Animation Controller in Unity is a system that manages the state machine for character animations. It defines how animations transition from one state to another based on conditions.

Animation Controller Script – An Animation Controller Script is a script written in a programming language such as C# that provides additional functionality or customization to the Animation Controller in Unity.

Animation Layer – Animation Layers in Unity are used for managing complex state machines for different body parts. For example, having a head look layer for turning the head with the camera direction, or having a lower-body layer for walking and jumping and an upper-body layer for throwing objects or shooting.

Animation State – Animation States represent individual animation clips or motions within an Animation Controller. These states define the specific animations a character can be in.

Blend Tree – A Blend Tree is a Unity mechanism that allows smooth transitions between animations by blending multiple animations based on input parameters, such as Speed or Direction.

Character Avatar – In Unity's Mecanim system, the Character Avatar refers to the digital representation of a character's skeletal structure, including bone hierarchy and rigging. It serves as the foundation for applying animations and controlling the character's movements within the game.

Mecanim – Mecanim is Unity's animation system, encompassing the Animation Controller, Animator component, and state machine. It provides a visual interface for designing complex character animations and facilitates the integration of character avatars, animation clips, and transitions.

Parameter – In Unity's Animation Controller, parameters are variables that can be accessed and assigned values. These parameters are often used to control the flow of the state machine or influence animations.

State Machine – In the context of Unity's Animation Controller, a State Machine is a computational model that defines the various states a character or object can inhabit. It manages transitions between states based on specified conditions, influencing the character's behavior and animations.

Strafe – In the context of character movement, strafing refers to moving sideways without changing the direction the character is facing.

Transition – Transitions in an Animation Controller define how the system moves from one animation state to another. Conditions, such as parameter values, trigger these transitions.

Trigger – In Unity's Animation Controller, a Trigger is a boolean parameter that is reset by the controller when consumed by a transition. It is often used to initiate specific animations.