

# Creating a 3D Characters Movement

## Getting Characters and Animations

### Introduction to Mixamo

Before we can start to work with a 3D characters we have to get them first. A great place to get 3D characters and their animation is [mixamo](https://mixamo.com). Mixamo allows you to Animate 3D characters with no 3D knowledge required. It allows you to Rapidly create, rig and animate unique characters for design projects.

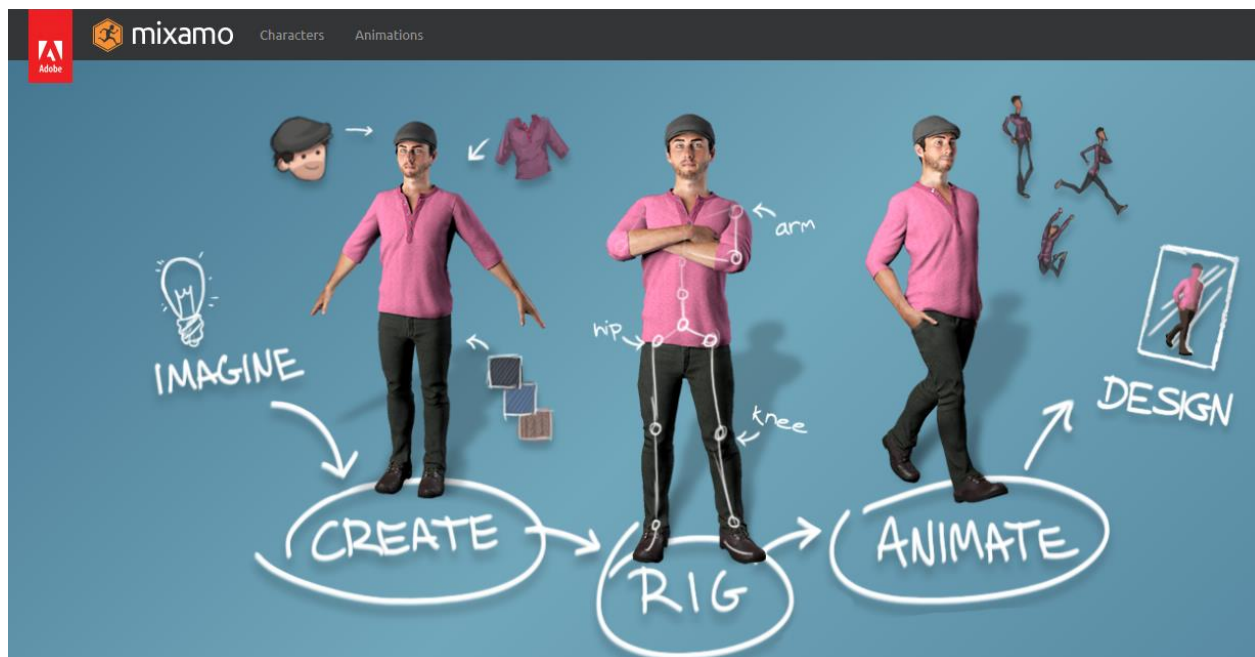


Figure 1 Mixamo Homepage

### Getting Characters

Mixamo allows you to download premaid and rigged characters or to upload your own and rig them based on its Automatic Rigging algorithm. For now we only want to use its chracters.

- Login to Mixamo (if you don't have an account make one).
- Click on the *Characters* tab in the top navigation bar in the website homepage.



Figure 2

- A page split into two parts will open. The right part will contain a lot of characters and the second part will contain a preview of the character (you can zoom in/out, rotate, pan, enable/disable skeleton view ).

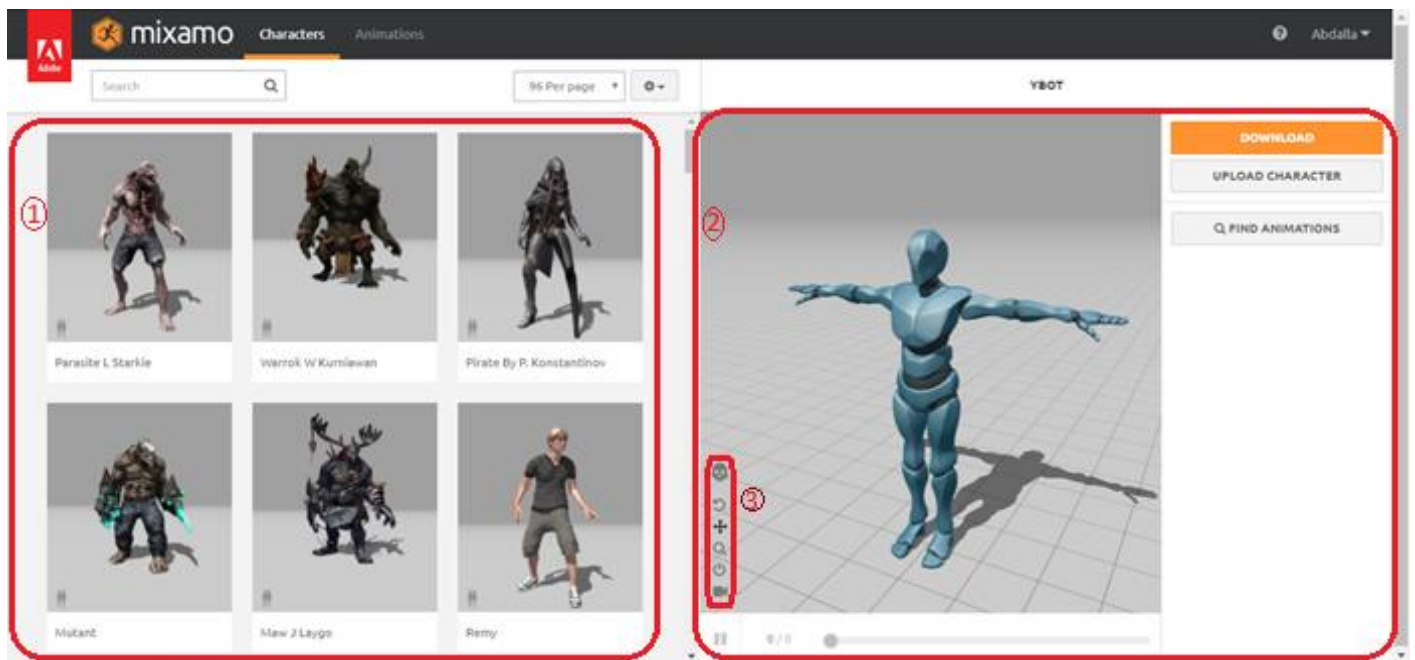


Figure 3 (1) is the character choose area. (2) character preview. (3) Tools for previewing the selected character.

- Search for “ybot” and click download.

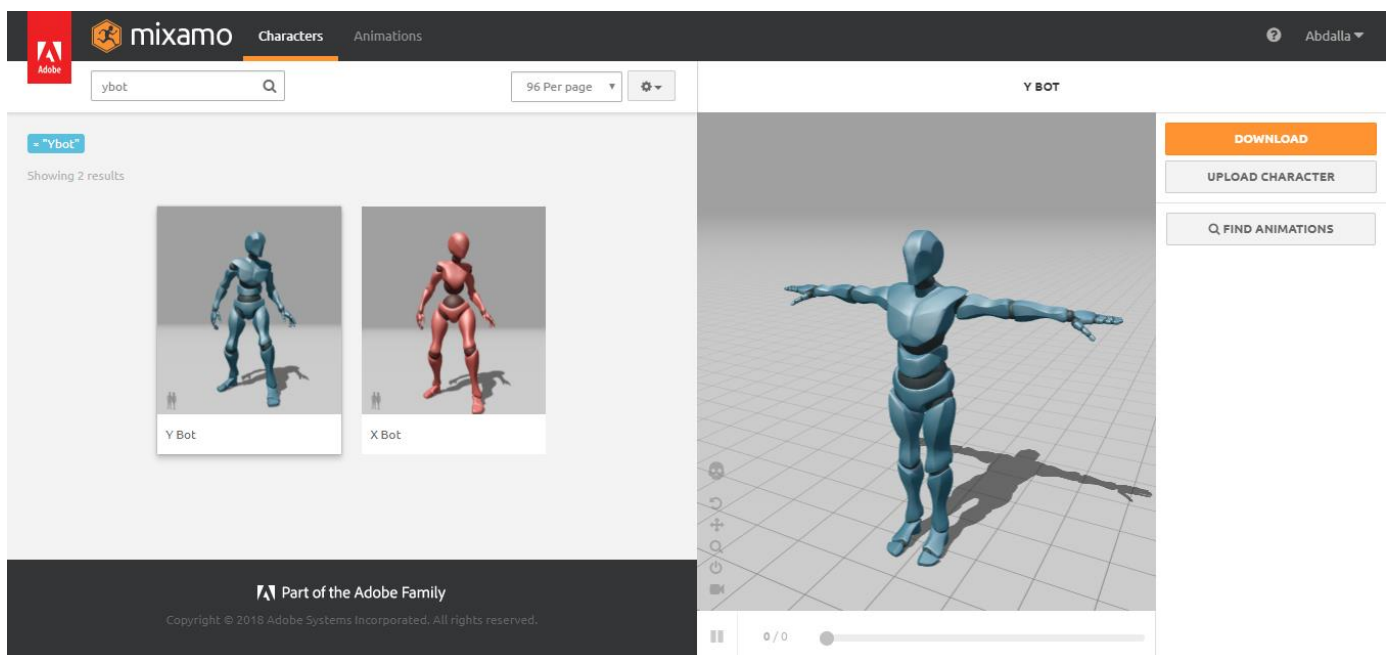
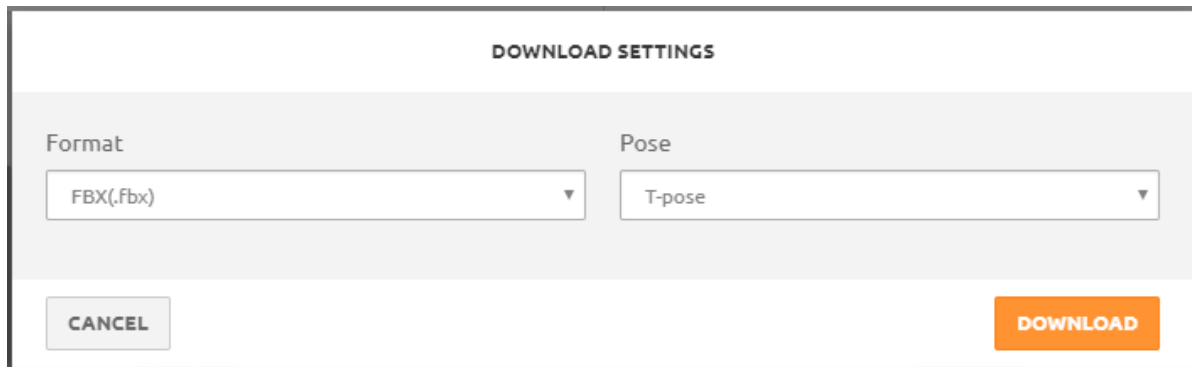


Figure 4

- A *Download Settings* dialog will open that contain the following:
  - Format  
The *extension* of the file. We will use “FBX for Unity(.fbx)”
  - Pose  
Character Pose. We will choose “T-pose”.



The image shows a 'DOWNLOAD SETTINGS' dialog box. It has a title bar at the top. Below the title bar, there are two dropdown menus. The first is labeled 'Format' and has 'FBX(.fbx)' selected. The second is labeled 'Pose' and has 'T-pose' selected. At the bottom of the dialog, there are two buttons: 'CANCEL' on the left and 'DOWNLOAD' on the right.

Figure 5 Character Download Settings

## T-pose

A **T-pose**, sometimes known as a **bind pose** or **reference pose**, is the default unanimated state of a model in 3D graphics. This pose is often with all of a model's various parts straightened out or flattened for ease of animation. For most characters, this results in a pose where the legs are straight and the arms are pointing sideways in a T-shape.

Source: <https://www.ssbwiki.com/T-pose>

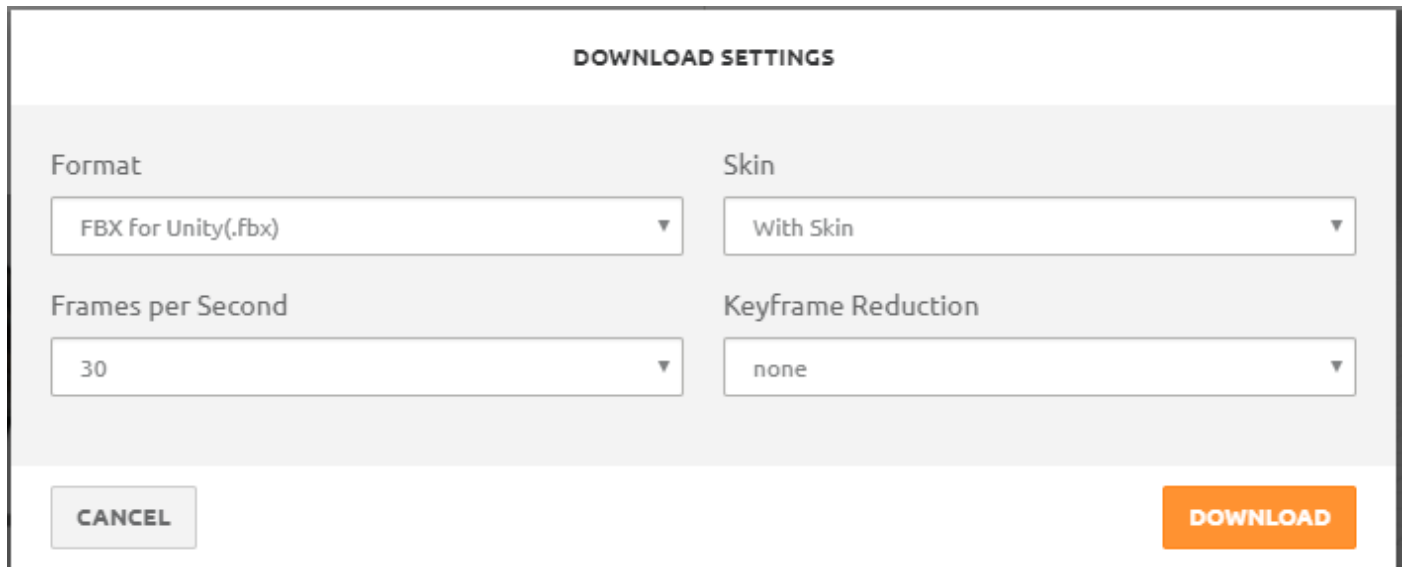
- A new file named “ybot.fbx” will be downloaded after clicking download.

### Getting Animations

The file we downloaded contains the 3D model only without any animation. We could attach an animation to it directly in Mixamo and download it with that animation. But I think it is better to download animation separately because this enable us to reuse them later easily.

To download Animations:

- Go to *Animations* tab
- Select the Animation you want  
After selecting the animation you can play with some of its attributes like Overdrive, Character Arm-Space, Trim and Mirror.
- When you click download a dialog will open which contain the following
  - Format  
We will chose “FBX for Unity(.fbx)” like before.
  - Skin  
Whether you want to download the character with the animation or not. We don’t want to download the character with the animation, so we will choose “Without Skin”.
  - Frames per Second  
Keep the default setting.
  - Keyframe Reduction  
Keep the default setting.



The screenshot shows a 'DOWNLOAD SETTINGS' dialog box. It has a light gray background. At the top, the title 'DOWNLOAD SETTINGS' is centered. Below the title, there are four dropdown menus arranged in a 2x2 grid. The first row contains 'Format' (set to 'FBX for Unity(.fbx)') and 'Skin' (set to 'With Skin'). The second row contains 'Frames per Second' (set to '30') and 'Keyframe Reduction' (set to 'none'). At the bottom of the dialog, there are two buttons: a gray 'CANCEL' button on the left and an orange 'DOWNLOAD' button on the right.

Figure 6 Mixamo animation download settings

Download the “Idle” animation for the T-Bot.  
A file named “ybot@Idle.fbx” will be downloaded.

In this tutorial we will use the following animations:

- Idle
- Walking
- Walking Backward
- Running
- Running Backward
- Jumping
- Climbing

### Creating the Project

Let’s create a new Unity project:

- Create a new *Unity project*, call it “Animation 01”.
- Create a new *Scene* call it “scene1”.
- Create a new *Terrain* call it “Ground”.
- Create the following folders in the Project View:
  - Animations
  - Models
  - Scripts
- Import the *ybot* model and put it in the “Models” folder.
- Download the animations and import them and place them in the “Animations” folder.
- Drag the *ybot* model to the *scene* and place it in the middle of our *Terrain*.
- Select the *ybot* GameObject in the *Hierarchy* and look at the *Inspector*.
- Change the position and orientation of the *camera* to view our character from the back.
- Add a **RigidBody** to our *ybot* GameObject and make sure that **Use Gravity** is checked.
  - Expand **Constraint** in the RigidBody and check **Freeze Rotation** around X, Y and Z axis.

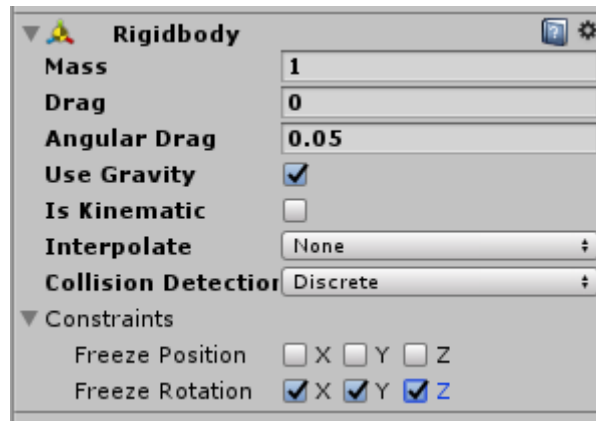
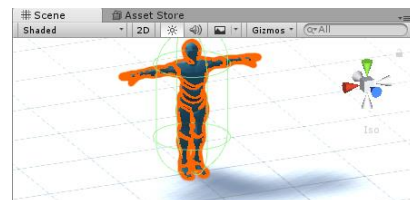


Figure 7

- Add a **Capsule Collider** component to our *ybot* GameObject  
You will notice a half of a green sphere is appeared.
  - Change the **Center Y** property to 1.
  - Change **Height** property to 2.



a



b

Figure 8 (a) after creating Capsule Colider. (b) after changing its Center and Height

- You will notice that our GameObject has an *Animator Component*, if it doesn't has one add one.
- Create a new *Animator Controller* and assign it to the *ybot* animator component *Controller* property.

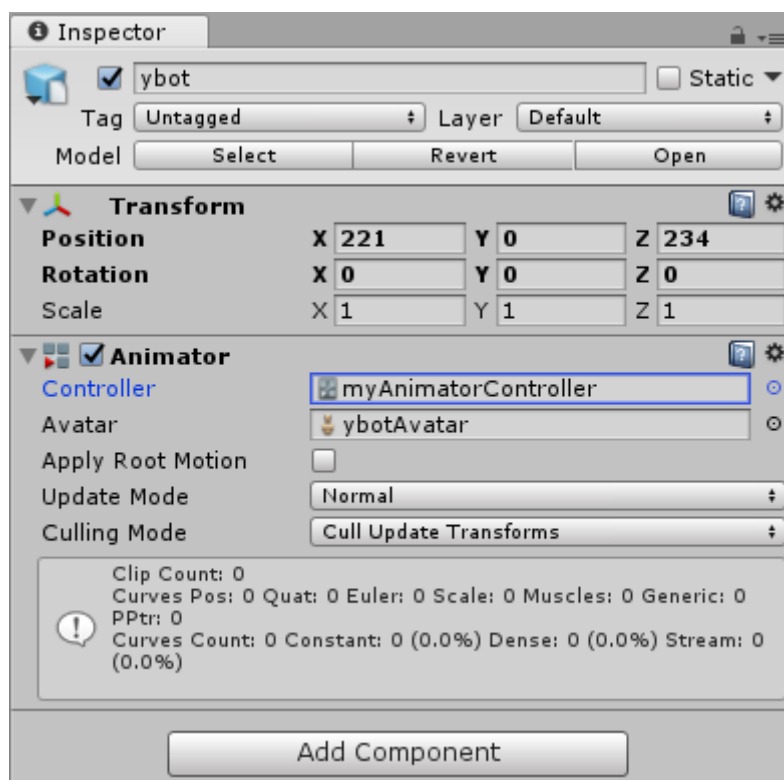


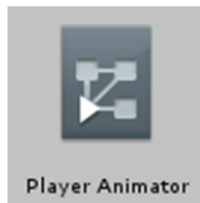
Figure 9

### Animator Controllers

An **Animator Controller** allows you to arrange and maintain a set of animations for a character or other animated Game Object.

The controller has references to the animation clips used within it and manages the various animation states and the transitions between them using a so-called **State Machine**, which could be thought of as a kind of flow-chart, or a simple program written in a visual programming language within Unity.

An Animator Controller asset is created within Unity and allows you to maintain a set of animations for a character or object.



*Figure 10 An Animator Controller Asset in the Project Folder*

Animator Controller assets are created from the *Assets menu*, or from the *Create menu* in the Project window.

- Let's open the *Animator Controller* by opening the *Animator Window* from the **Window** menu.

### The Animator Window

It allows you to create, view and modify Animator Controller assets.

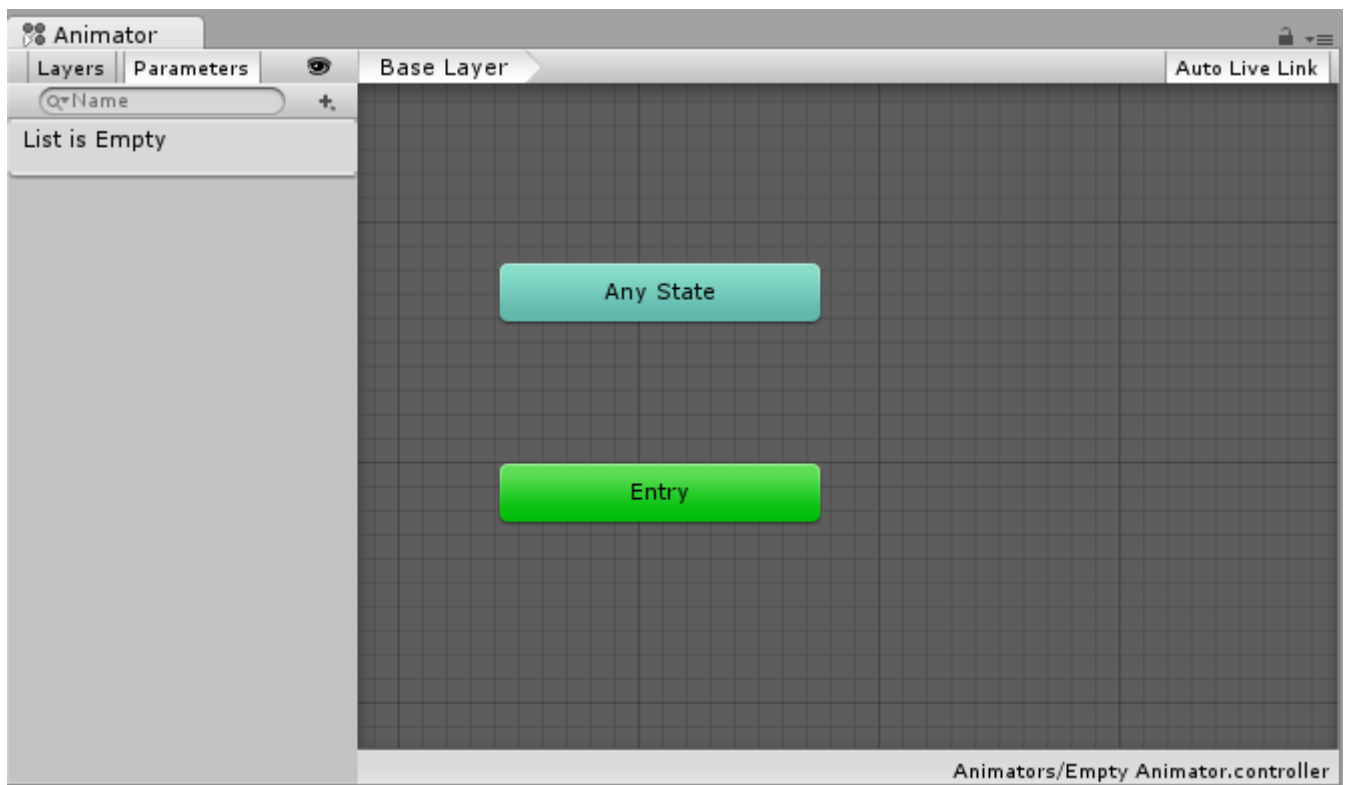


Figure 11 The Animator Window showing a new empty Animator Controller asset

The Animator window has two main sections:

- The main gridded layout area.  
The main section with the dark grey grid is the layout area. You can use this area to create, arrange and connect states in your Animator Controller.
  - You can right-click on the grid to create a new state node.
  - Use the middle mouse button or Alt/Option drag to pan the view around.
  - Click to select state nodes to edit them and click & drag state nodes to rearrange the layout of your state machine.
- The left-hand Layers & Parameters pane.  
The left-hand pane can be switched between Parameters view and Layers view.
  - The *parameters view* allows you to create, view and edit the **Animator Controller Parameters**. These are variables you define that act as inputs into the state machine.
  - *Layers view* allows you to create, view and edit layers within your Animator Controller. This allows you to have multiple layers of animation within a single animation controller working at the same time, each controlled by a separate state machine.
    - A common use of this is to have a separate layer playing upper-body animations over a base layer that controls the general movement animations for a character.

## Animation State Machines

It is common for a character or other animated Game Object to have several different animations that correspond to different actions it can perform in the game. For example:

- A character may breathe or sway slightly while idle, walk when commanded to and raise its arms in panic as it falls from a platform.
- A door may have animations for opening, closing, getting jammed, and being broken open.

Mecanim uses a visual layout system similar to a flow-chart, to represent a **state machine** to enable you to control and sequence the animation clips that you want to use on your character or object.

### What's "Mecanim"?

Mecanim was the name of the animation software that we integrated into Unity. Early in the 4.x series of Unity, its abilities were tied specifically to humanoid character animation and it had many features which were uniquely suited for that purpose, and it was separate to our old (now legacy) integrated animation system.

Mecanim integrated humanoid animation retargeting, muscle control, and the state machine system. The name "Mecanim" comes from the French word "Mec" meaning "Guy". Since Mecanim operated only with humanoid characters, our legacy animation system was still required for animating non-humanoid characters and other keyframe-based animation of gameobjects within Unity.

Since then however, we've developed and expanded Mecanim and integrated it with the rest of our animation system so that it can be used for all aspects of animation within your project - so there is a less clear definition where "Mecanim" ends and the rest of the animation system begins. For this reason, you'll still see references in our documentation and throughout our community to "Mecanim" which has now simply come to mean our main animation system.

Source: [Animation FAQ](#)

## State Machine Basics

The basic idea is that a character is engaged in some kind of action at any given time. The actions available will depend on the type of gameplay but typical actions include things like *idling*, *walking*, *running*, *jumping*, etc. These actions are referred to as **states**, in the sense that the character is in a "state" where it is walking, idling or whatever.

In general, the character will have restrictions on the next state it can go to rather than being able to switch immediately from any state to any other. For example, a running jump can only be taken when the character is already running and not when it is at a standstill, so it should never switch straight from the idle state to the running jump state. The options for the next state that a character can enter from its current state are referred to as **state transitions**.

Taken together, the set of states, the set of transitions and the variable to remember the current state form a **State Machine**.

The states and transitions of a state machine can be represented using a graph diagram, where the nodes represent the states and the arcs (arrows between nodes) represent the transitions.

The importance of state machines for animation is that they can be designed and updated quite easily with relatively little coding. Each state has a Motion associated with it that will play whenever the machine is in that state. This



enables an animator or designer to define the possible sequences of character actions and animations without being concerned about how the code will work.

### State Machines

Unity's Animation State Machines provide a way to overview all of the animation clips related to a particular character and allow various events in the game (for example user input) to trigger different animations.

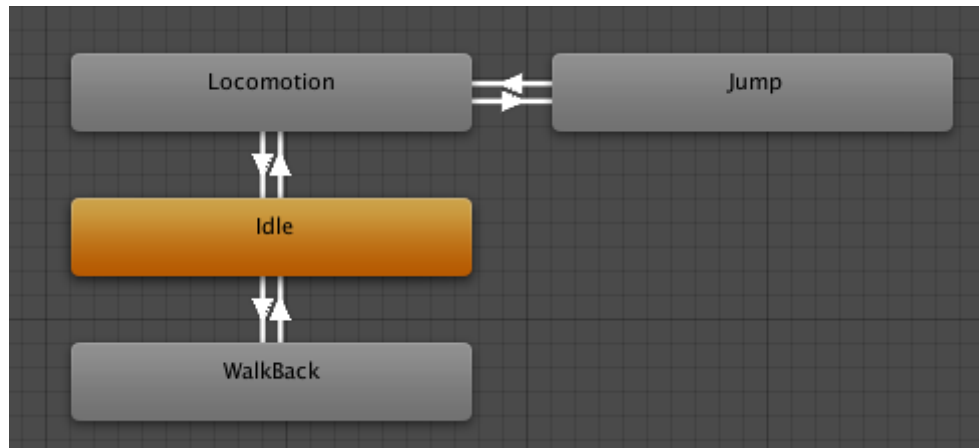


Figure 12

State Machines consist of **States**, **Transitions** and **Events** and smaller Sub-State Machines can be used as components in larger machines.

### Animation States

**Animation States** are the basic building blocks of an *Animation State Machine*. Each state contains an individual animation sequence (or blend tree) which will play while the character is in that state. When an event in the game triggers a state transition, the character will be left in a new state whose animation sequence will then take over.

The default state, displayed in brown, is the state that the machine will be in when it is first activated. You can change the default state, if necessary, by right-clicking on another state and selecting **Set As Default** from the context menu.

Each view in the animator window (Animation State Machine) has an *Entry* and *Exit* node (state). These are used during State Machine Transitions.

- The *Entry node (Entry State)* is used when transitioning into a state machine.
  - The *entry node* will be evaluated and will branch to the destination state according to the conditions set. In this way the entry node can control which state the state machine begins in, by evaluating the state of your parameters when the state machine begins.
  - Because state machines always have a *default state*, there will always be a *default transition* branching from the entry node to the default state.

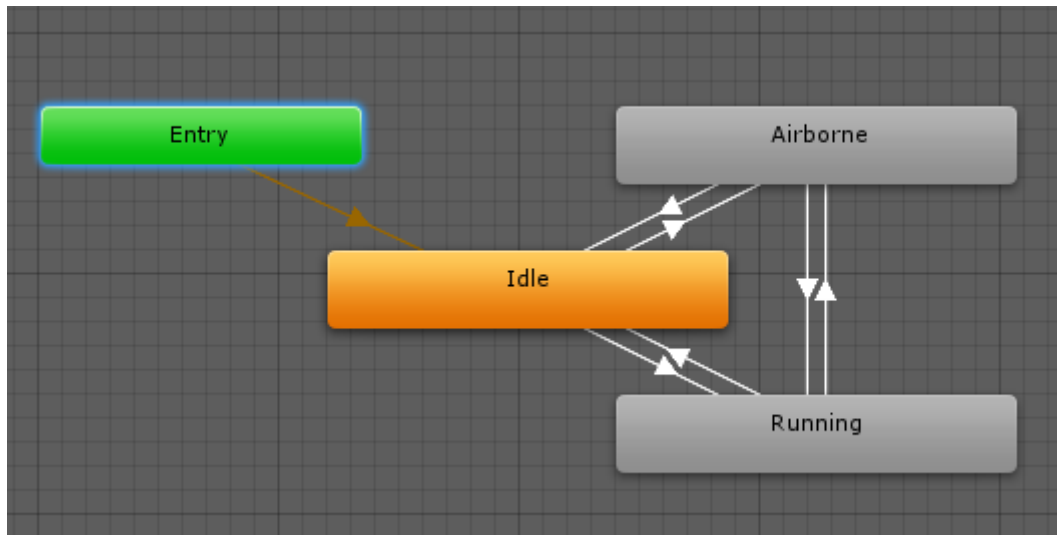


Figure 13 An entry node with a single default entry transition

- The *Exit node* (Exit State) is used to indicate that a state machine should exit.

### Adding Animation States

Let us continue our Unity Project. We want to add new states that will represent the actions of our character. We will start by the following actions/states:

- Idle
- Walking Forward
- Walking Backward
- Jumping

To create a new Animation State (I will use *State* to refer to Animation State) :

- Right Click on the gridded *Layout Area*, choose **Create State** then choose **Empty**. A new state will be added to the Layout Area.

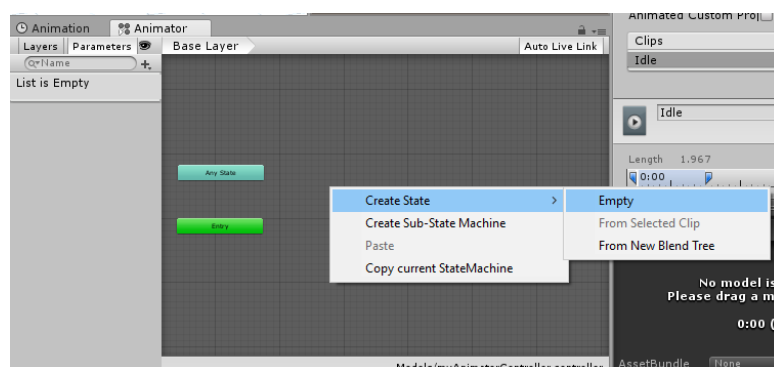


Figure 14 Creating a new Animation State

Note that the first state you create will be the **Default State**, with an orange color. And a default transition will be created from the *Entry* state to the newly created state.

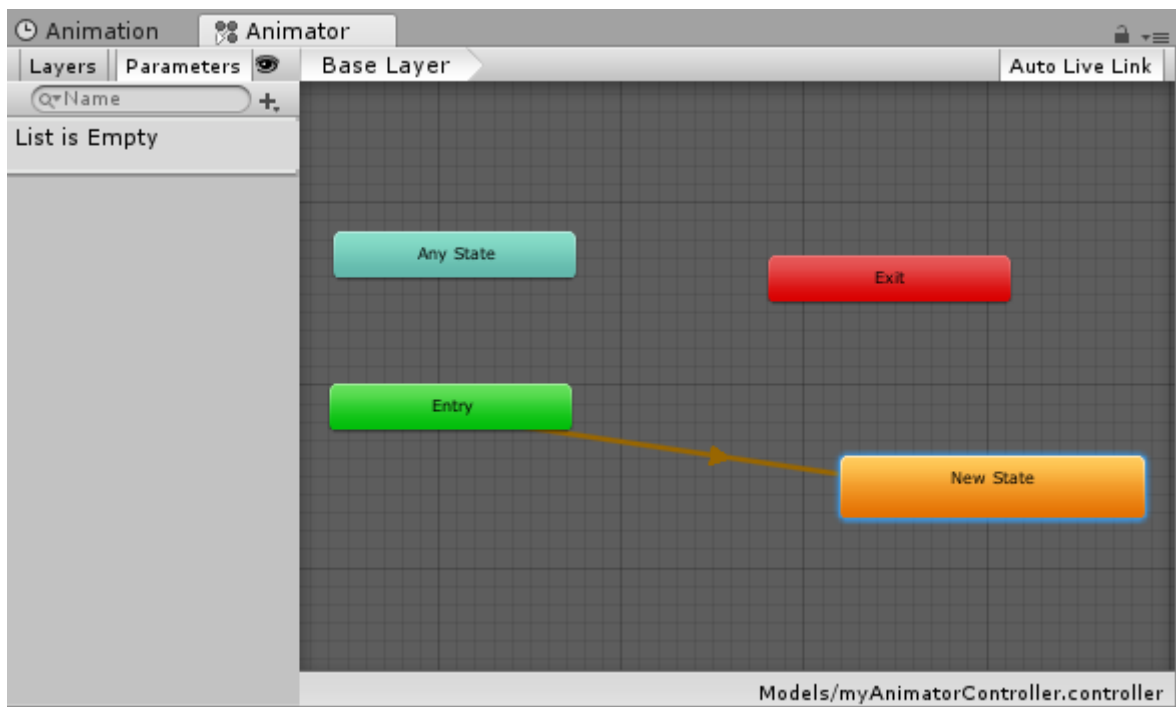


Figure 15 After creating the first state.

A new state can be added by right-clicking on an empty space in the **Animator Controller Window** and selecting **Create State->Empty** from the context menu.

Alternatively, you can drag an animation into the Animator Controller Window to create a state containing that animation. (Note that you can only drag Mecanim animations into the Controller - non-Mecanim animations will be rejected.)

- Select the newly created state, named “New State” then go to the inspector to view state properties.

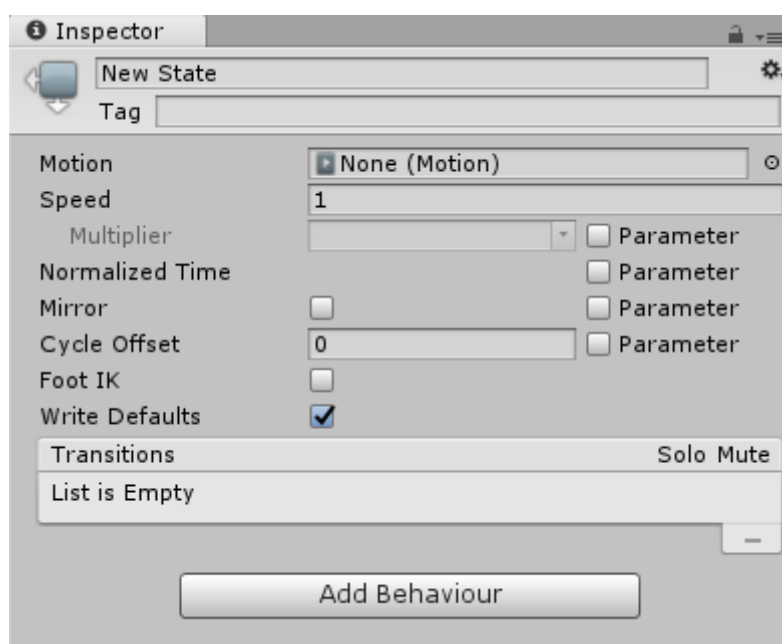


Figure 16 State properties

- Change the name of the *State* to “Idle”.
- Change the *Motion* property to reference the Idle animation.  
It should be in **Assets/Animations/ybot@Idle**.

## State Properties

<i><b>Property:</b></i>	<i><b>Function:</b></i>
<b>Speed</b>	The default speed of the animation
<b>Motion</b>	The animation clip assigned to this state
<b>Foot IK</b>	Should Foot IK be respected for this state. Applicable to humanoid animations.
<b>Write Defaults</b>	Whether or not the AnimatorStates writes back the default values for properties that are not animated by its Motion.
<b>Mirror</b>	Should the state be mirrored. This is only applicable to humanoid animations.
<b>Transitions</b>	The list of transitions originating from this state

For now we are concerned only with **Motion** and **Transitions** properties.

- Add the *WalkingForward*, *WalkingBackward* and *Jumping* states use the same way.  
Your state machine should include the states as in the image below.

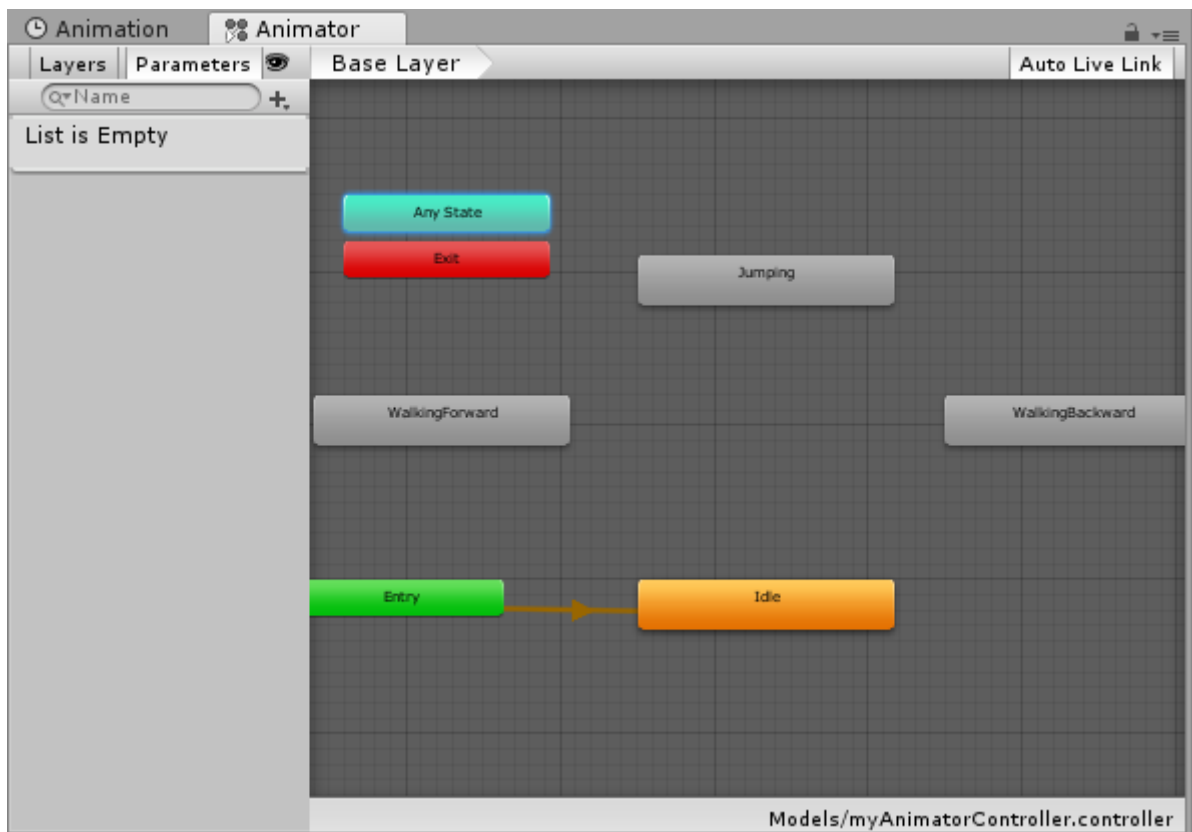


Figure 17

- Play the game and observe what happen in the Animator Window.  
The idle animation will played once and then stop. You will probably want it to continue unless another action happened. That is, we must Loop the animation clip.



Figure 18

- To loop an Animation Clip:
  - Select the Animation File from the Project Window.  
Select **Assets/Animations/ybot@Idle**.
  - Go the Inspector and scroll down till you see **Loop Time**.
  - Check **Loop Time** and check **Loop Pose**.
  - Scroll down and click **Apply** button.
  - Try to play the game again and see the difference.

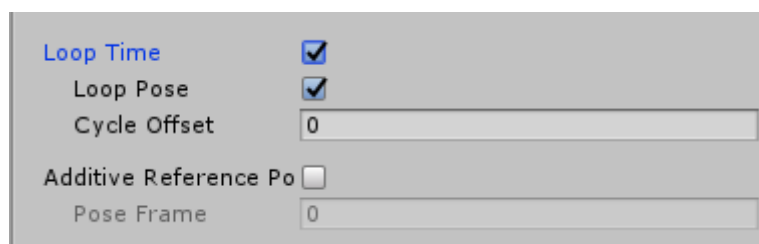


Figure 19

- We want our Character to be able to change its action from *Idle* to any of the other 3 actions (Walking Forward, Walking Backward and Jumping) and vice versa. To do that we should add **Transitions** from and to the *Idle* state. And we will do that to the other three states too.

### Animation transitions

Animation transitions allow the state machine to switch or blend from one animation state to another. Transitions define not only how long the blend between states should take, but also under what conditions they should activate.

You can set up a transition to occur only when certain conditions are true. To set up these conditions, specify values of **parameters** in the Animator Controller.

To add a **Transition**:

- Select the *State* you want to make the transition from, **Right Click** on it and Select **Make Transition**. A white arrow will follow the mouse.
- Drag the mouse to the state you want the transition to go to and Click with the mouse. The arrow will freeze on this state and by that you have finished making the transition.

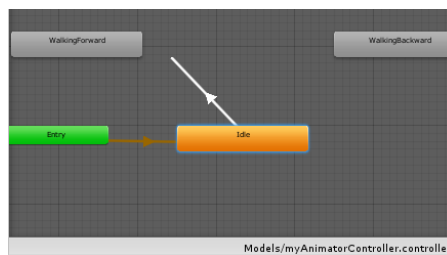


Figure 20

### Transition properties

When you click on a transition, you can view its properties in the inspector.

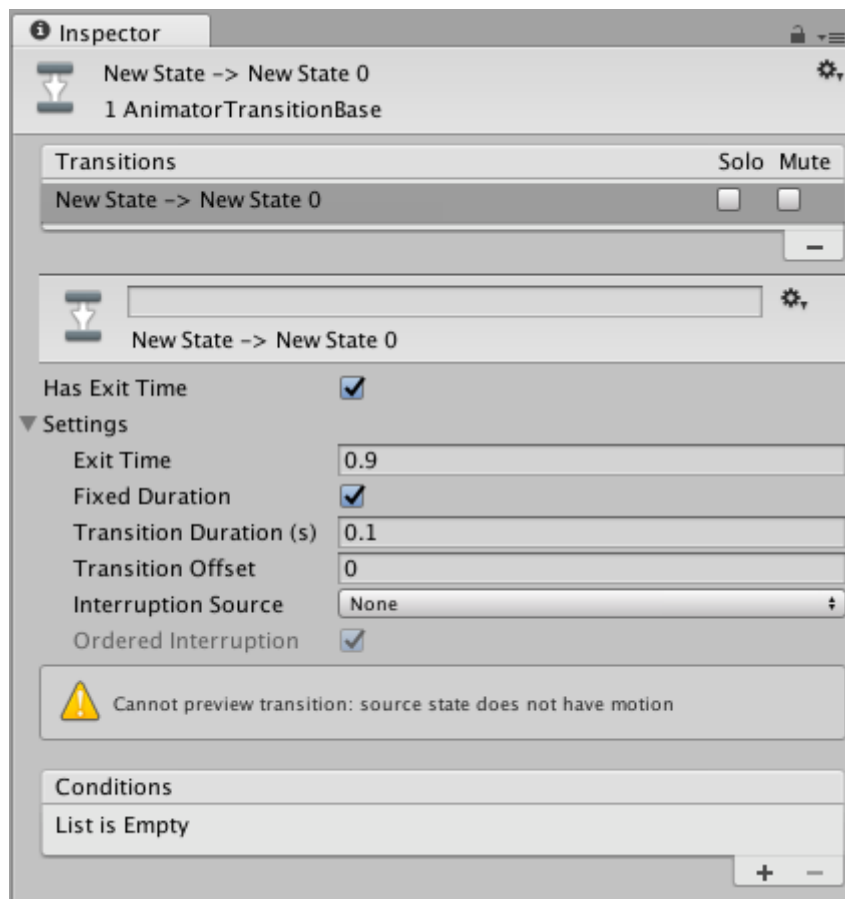


Figure 21 Transition Properties

Property	Function
<b>Has Exit Time</b>	<b>Exit Time</b> is a special transition that doesn't rely on a parameter. Instead, it relies on the normalized time of the state. Check to make the transition happen at the specific time specified in <b>Exit Time</b> .
<b>Settings</b>	Fold-out menu containing detailed transition settings as below.
<b>Exit Time</b>	<p>If <b>Has Exit Time</b> is checked, this value represents the exact time at which the transition can take effect. This is represented in normalized time (for example, an exit time of 0.75 means that on the first frame where 75% of the animation has played, the <b>Exit Time</b> condition is true). On the next frame, the condition is false.</p> <p>For looped animations, transitions with exit times smaller than 1 are evaluated every loop, so you can use this to time your transition with the proper timing in the animation every loop.</p> <p>Transitions with an <b>Exit Time</b> greater than 1 are evaluated only once, so they can be used to exit at a specific time after a fixed number of loops. For example, a transition with an exit time of 3.5 are evaluated once, after three and a half loops.</p>
<b>Fixed Duration</b>	If the <b>Fixed Duration</b> box is checked, the transition time is interpreted in seconds. If the <b>Fixed Duration</b> box is not checked, the transition time is interpreted as a fraction of the normalized time of the source state.
<b>Transition Duration</b>	The duration of the transition, in normalized time or seconds depending on the <b>Fixed Duration</b> mode, relative to the current state's duration. This is visualized in the transition graph as the portion between the two blue markers.
<b>Transition Offset</b>	The offset of the time to begin playing in the destination state which is transitioned to. For example, a value of 0.5 means the target state begins playing at 50% of the way through its own timeline.
<b>Conditions</b>	<p>A transition can have a single condition, multiple conditions, or no conditions at all. If your transition has no conditions, the Unity Editor only considers the <b>Exit Time</b>, and the transition occurs when the exit time is reached. If your transition has one or more conditions, the conditions must all be met before the transition is triggered.</p> <p><b>A condition consists of:</b></p> <ul style="list-style-type: none"> <li>- An event parameter (the value considered in the condition).</li> <li>- A conditional predicate (if needed, for example, 'less than' or 'greater than' for floats).</li> <li>- A parameter value (if needed).</li> </ul> <p>If you have <b>Has Exit Time</b> selected for the transition and have one or more conditions, note that the Unity Editor considers whether the conditions are true after the <b>Exit Time</b>. This allows you to ensure that your transition occurs during a certain portion of the animation.</p>

### Transition graph

To manually adjust the **Transition settings**, you can either enter numbers directly into the fields or use the transition graph. The transition graph modifies the values of the Transition settings when the visual elements are manipulated.

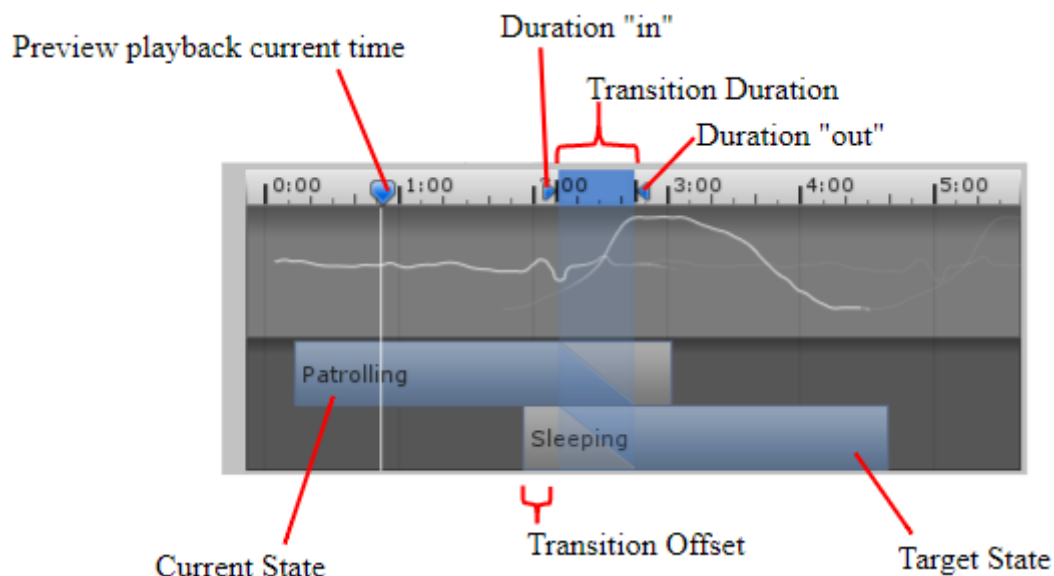


Figure 22 The Transition settings and graph as shown in the Inspector

Change the transition properties in the graph view using the following directions:

- Drag the Duration “out” marker to change the Duration of the transition.
- Drag the Duration “in” marker to change the duration of the transition and the Exit Time.
- Drag the target state to adjust the Transition Offset.
- Drag the preview playback marker to scrub through the animation blend in the preview window at the bottom of the Inspector.

Let us continue our work...

- Create a transition from **Idle** to **WalkingForward**.  
Idle -> WalkingForward
- Play the game and see what happens.  
The game began with the **Idle** animation then go to the **WalkingForward** animation and then stop.

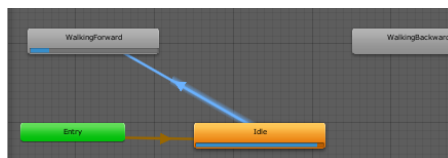


Figure 23

- Select the transition to view its properties in the inspector.



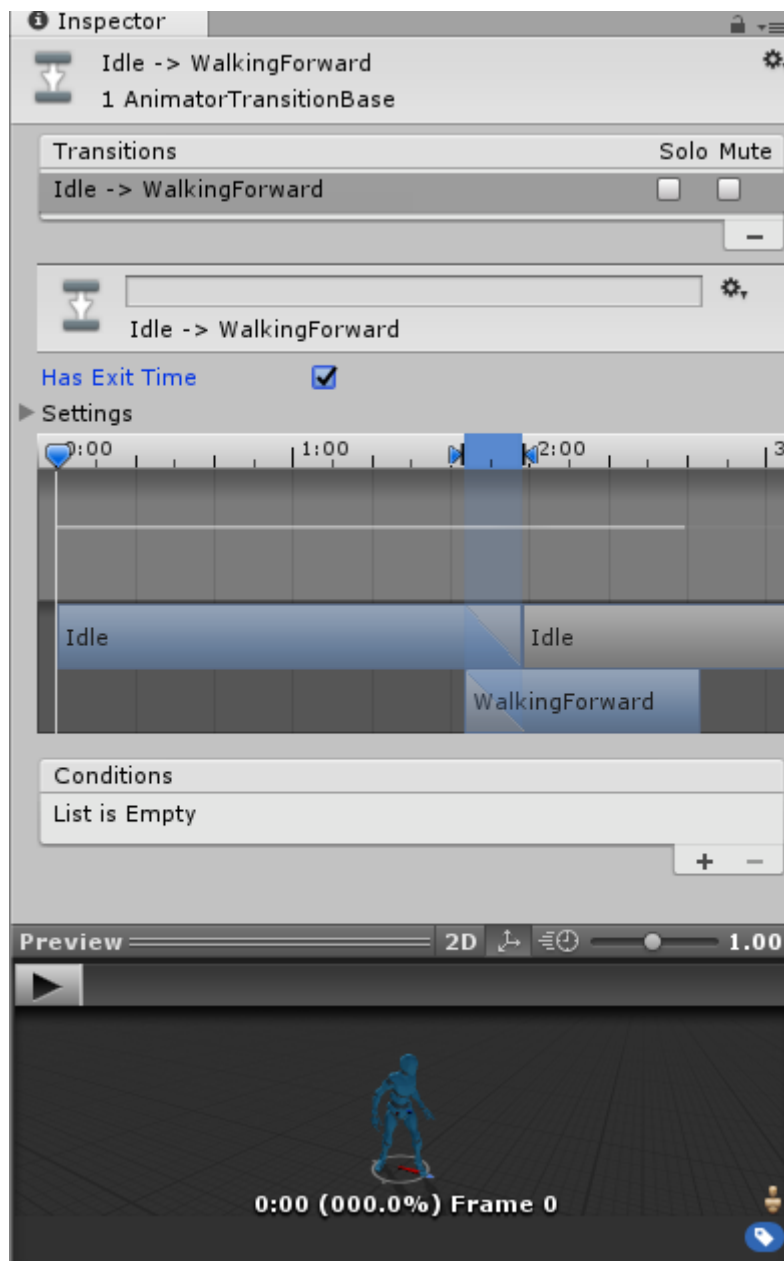


Figure 24

- Uncheck the **Has Exit Time** property. Play the game again and notice the difference. Now only the **idle** animation plays.

Remember that:

**Has Exit Time** is a special transition that doesn't rely on a parameter. Instead, it relies on the normalized time of the state. Check to make the transition happen at the specific time specified in **Exit Time**.

That means that if you want your animation to rely only on the parameter you created, disable the **Has Exit Time**.

- Create the following transitions:

- WalkingForward to Idle
- Idle to WalkingBackward
- WalkingBackward to Idle
- Idle to Jumping
- Jumping to Idle
- Uncheck Has Exit Time from all transitions.
- The State Machine should look like the image below after making the transitions above.

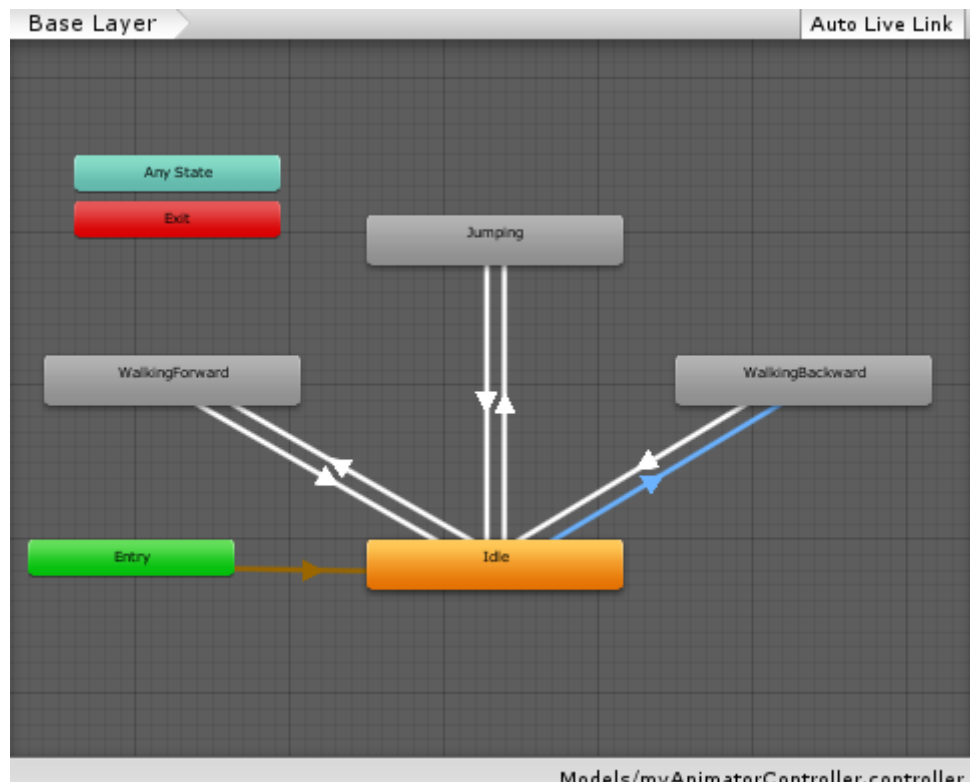


Figure 25

To be able to control change of states and when transitions should start we need to use **Conditions**. To make condition we need to make **Animation Parameters**.

#### Animation Parameters

**Animation Parameters** are variables that are defined within an *Animator Controller* that can be accessed and assigned values from scripts. This is how a script can control or affect the flow of the state machine.

Default parameter values can be set up using the Parameters section of the Animator window, selectable in the top right corner of the Animator window.

They can be of four basic types:

- **Int** - an integer (whole number)
- **Float** - a number with a fractional part
- **Bool** - true or false value (represented by a checkbox)
- **Trigger** - a boolean parameter that is reset by the controller when consumed by a transition (represented by a circle button)

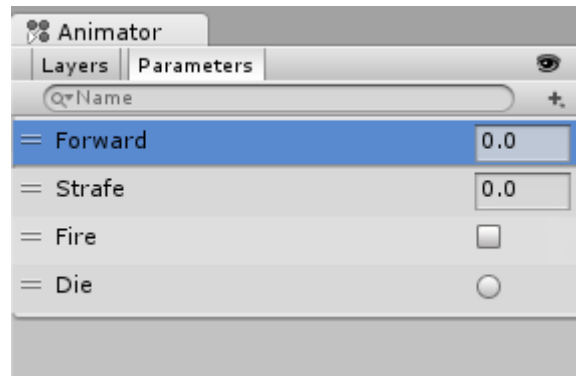


Figure 26 Animation Parameters

Parameters can be assigned values from a script using functions in the Animator class: `SetFloat`, `SetInt`, `SetBool`, `SetTrigger` and `ResetTrigger`.

To create an **Animation Property**:

- Open the **Parameters** tab in the Animator window.
- Click the **plus sign**
- Choose the **Type** of the property.
- A new property will be created, and you will be able to change its name.

To control our transition in the state machine we will need to create the following **Animation Properties**:

- `isWalkingForward` of type Boolean.
- `isWalkingBackward` of type Boolean.
- `isIdle` of type Boolean.
- `isJumping` of type trigger.

The values of these **Animation Properties** will be set from the **script** that we will create later.

Let's modify our Transitions and add conditions to them.

We want our Transition from **Idle** to **WalkingForward** to work when `isWalkingForward` is **true**. So we will add this to the conditions of the Transition.

- Select the Transition from **Idle** to **WalkingForward**.
- Go to the **Inspector > Condition List** and click the plus sign.
- A new **Condition** will be added to the **Condition List**.  
A condition consists of:
  - An event parameter (the value considered in the condition).
  - A conditional predicate (if needed, for example, 'less than' or 'greater than' for floats).
  - A parameter value (if needed).
- Choose `isWalkingForward` and choose its value to be **true**.
- Select the Transition from **WalkingForward** to **Idle**.
- Make a new condition in the Transition and choose the **parameter** to be `isIdle` and its value to be **true**.
- Add conditions to other states in a similar way.
  - Idle to **WalkingBackward**  
Add Condition: `isWalkingBackward` equals to **true**.
  - **WalkingBackward** to **Idle**  
Add Condition: `isIdle` equals to **true**.

- Idle to Jumping  
Add Condition: `isJumping`.
- Jumping to Idle  
Add Condition: `isIdle` equals to `true`.

## Movement Script

Now its time to write some code to make the animation play.

- Go to **Assets/Scripts** folder.
- Add a new Script call it "MovementController".
- Open the Script.
- Create the following properties as in the code snippet below.

```
public bool isGrounded;
private float speed;
public float rotSpeed;
public float jumpHeight;
//walk speed
private float w_speed = 0.05f;;
//rotation speed
private float rot_speed = 1.0f;
Rigidbody rb;
Animator anim;
```

Code Snippet 1

- `isGrounded` is used to detect when we are in the ground and when we reach it from jumping. Its initial value will be set to `true` in the `Start` method.
- `speed` will be used to differentiate between normal walking and running.
- `w_speed` and `rot_speed` are the values we will use for walking and rotation.
- `rb` and `anim` will be reference to `Rigidbody` component and `Animator`.
- Initialization is done in the `Start` method as follow:

```
// Use this for initialization
void Start()
{
    rb = GetComponent<Rigidbody>();
    anim = GetComponent<Animator>();
    isGrounded = true; //indicate that we are in the ground
}
```

Code Snippet 2

- We will make a function (`MovementControl`)that will handle the changes in of our **Animator Parameters**. Refer to the Code Snippet below.  
The function work as follow:
  - We pass a `state` to the function. This state refers to the action that is captured in the `Update` method base on the user input.
  - Based on the State we set the appropriate value of the `Animator Parameter`.
  - Based on that change a transition may be fired in our state machine.

```
void movementControl(string state)
{
    switch (state)
    {
        case "WalkingForward":
            anim.SetBool("isWalkingForward", true);
            anim.SetBool("isWalkingBackward", false);
            anim.SetBool("isIdle", false);
            break;
        case "WalkingBackward":
            anim.SetBool("isWalkingForward", false);
            anim.SetBool("isWalkingBackward", true);
            anim.SetBool("isIdle", false);
            break;
        case "idle":
            anim.SetBool("isWalkingForward", false);
            anim.SetBool("isWalkingBackward", false);
            anim.SetBool("isIdle", true);
            break;
    }
}
```

*Code Snippet 3*

Let us look on our Update method.

- We need to handle forward/backward movement.  
Forward is indicated by using the “W” key, and Backward is indicated by using the “S” key.
- We need to handle rotation of the character (going right and left).  
The “A” and “D” keys.
- We need to handle jumping.  
The “Space” button

The Update method is on the code snippet below.

```
// Update is called once per frame
void Update()
{
    if (isGrounded)
    {
        //moving forward and backward
        if (Input.GetKey(KeyCode.W))
        {
            speed = w_speed;
            movementCtrl("WalkingForward");
        }
        else if (Input.GetKey(KeyCode.S))
        {
            speed = w_speed;
            movementCtrl("WalkingBackward");
        }
        else
        {
            movementCtrl("idle");
        }
        //moving right and left
        if (Input.GetKey(KeyCode.A))
        {
            rotSpeed = rot_speed;
        }
        else if (Input.GetKey(KeyCode.D))
        {
            rotSpeed = rot_speed;
        }
        else
        {
            rotSpeed = 0;
        }
    }

    var z = Input.GetAxis("Vertical") * speed;
    var y = Input.GetAxis("Horizontal") * rotSpeed;
    transform.Translate(0, 0, z);
    transform.Rotate(0, y, 0);
    //jumping function
    if (Input.GetKeyDown(KeyCode.Space) && isGrounded == true)
    {
        anim.SetTrigger("isJumping");
        rb.AddForce(0, jumpHeight * Time.deltaTime, 0);
        isGrounded = false;
    }
}
```

Code Snippet 4

The Update method work as follow:

- We first check if we are in the ground or not by checking the `isGrounded` field.
- If `isGrounded` is `true`, then we do the following:
  - Check for **user input** ("W" and "S") to handle waking forward and backward. Based on that we have three cases:
    - Case 1: "W" is pressed. So, we need to

- Play the *walk forward animation*.
- Assign a walking speed ( `w_speed` ) to our `speed` field.
- The changing of the **Animation Parameter** is done in the `MovementControl` function.
- Changing in the Animation Parameter cause the transition from current state to the `WalkingForward` state to be fired.
- Case 2: “S” is pressed. So, we need to
  - Play the *walk backward animation*.
  - Assign a walking speed ( `w_speed` ) to our `speed` field.
  - The changing of the **Animation Parameter** is done in the `MovementControl` function.
  - Changing in the Animation Parameter cause the transition from current state to the `WalkingBackward` state to be fired.
- Case 3: We are idle or just rotating
  - Play the idle animation.
  - We can also reset `speed` to zero.
  - The changing of the **Animation Parameter** is done in the `MovementControl` function.
- Check for **user input** (“A” and “D”) to handle turning right and left. Based on that we have three cases like the above three cases but in this part, we change the rotation speed.
- The previous part handle the animation playing we now need to make the actual move (forward/ backward) and the actual rotation.  
To achieve that we rely on the `Input.GetAxis( "Vertical" )` method. And on the `Translate` and `Rotate` methods.

## Virtual Axes

Virtual axes and buttons can be created in the **Input Manager**.

From scripts, all virtual axes are accessed by their name.

Every project has the following default input axes when it's created:

- **Horizontal** and **Vertical** are mapped to w, a, s, d and the arrow keys.
- **Fire1**, **Fire2**, **Fire3** are mapped to Control, Option (Alt), and Command, respectively.
- **Mouse X** and **Mouse Y** are mapped to the delta of mouse movement.
- **Window Shake X** and **Window Shake Y** is mapped to the movement of the window.

For more information about how it works refer to the [GetAxis](#) tutorial.

You can also refer to the [Input Manager page](#) in the documentation.

- The next part handles the Jumping part.
  - If the “Space” button is pressed and `isGrounded` is true do the following:
    - Set `isJumping` trigger using `SetTrigger` method to play the animation.
    - Add force in the *positive Y* direction to make the `GameObject` jump.
    - Set `isGrounded` to false, to indicate that the object is not in the Ground.
- The remaining part is the part that set `isGrounded` to true when the `GameObject` land.  
This can be achieved by **OnCollisionEnter** method, see the code snippet below.

```
void OnCollisionEnter()  
{  
    isGrounded = true;  
}
```

*Code Snippet 5*

## Colliders

**Collider** components define the shape of an object for the purposes of physical collisions. A collider, which is invisible, need not be the exact same shape as the object's mesh and in fact, a rough approximation is often more efficient and indistinguishable in gameplay.

## Script actions taken on collision

When collisions occur, the physics engine calls functions with specific names on any scripts attached to the objects involved. You can place any code you like in these functions to respond to the collision event. For example, you might play a crash sound effect when a car bumps into an obstacle.

On the first physics update where the collision is detected, the `OnCollisionEnter` function is called. During updates where contact is maintained, `OnCollisionStay` is called and finally, `OnCollisionExit` indicates that contact has been broken.

For more information refer to the following links:

- [Colliders](#) in Unity Documentation.
- [Collider](#) in Script Reference.

Now try to play the game, use the A, W, S and D buttons to try the movement animation.

You will notice that the walking forward animation stops after a while from pressing "W" button this mean that **Loop Time** is not checked in the Animation Properties.

You will also notice some problems with animation:

- When changing from **WalkingForward** to **WalkingBackward**.
- When **WalkingForward** or Backward and Jump.

This happen because there is no transition form for example **walkingForward** to **walkingBackward**.

The following image show the State Machine after adding more transitions.



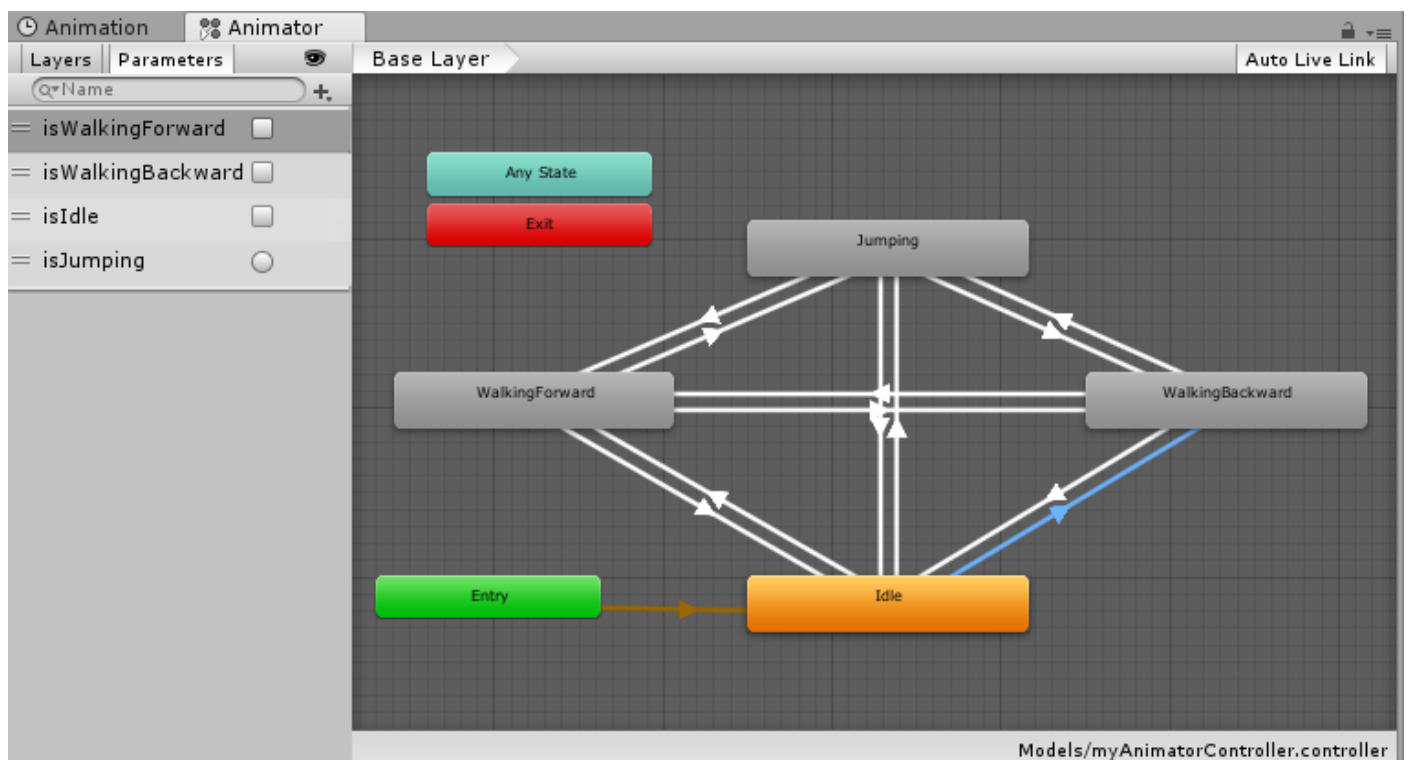


Figure 27