Contents

Animazing

~ How To Use ~

1. Introduction

Thank you very much for purchasing Animazing - a revolutionarily simple way to manage a complex amount of animations, states and behaviors!

For an example of use, please see the scene:

Old Odin / Animazing / Scenes / animazing.unity

and the script:

Old Odin / Animazing / Scripts / Sample / SimpleThirdPersonController.cs.

Below, we briefly explain what the main features of Animazing are and how to get the most out of them for your game.

2. The Priority System

One of the main setbacks when dealing with a complex character is managing the conflict between behaviors. For example, if the player can walk with W, attack with E and suffer damage when touching an enemy, what animation will prevail if all these conditions are present at the SAME time?

There are at least two traditional approaches to this problem:

a) User several chained IFs and ELSEs and check various conditions, for example: "if you are not suffering damage, if you are not attacking, and hit W, then you can walk". Which can become very complicated and confusing in characters with many different competing behaviors.

b) Use a state machine like Animator Controller. While this can be visually more intuitive, it can still be complex, laborious and even too limited in some contexts. In a sense, this just shifted the problem from one place to another: from the countless IFs to the countless Transitions between states.

In comparison, to decide which animation will take priority over competitors, Animazing simply asks the developer to play the animation and enter a number representing the animation's priority.

animazing.**Pla**y (animationClip, priorityNumber)

The higher the number, the higher the priority. For example, if walking has priority 1, attacking has priority 2, and taking damage has priority 3, if the 3 animations try to be played at the same time, the only animation that will prevail will be the damage animation, because it has a higher priority than the ones two others.

Likewise, if the player is walking (priority 1), and presses punch (priority 2), the walking animation will give way to the attack animation naturally.

You do not need to do any complex checks on various conditions. You do not need to define smooth transitions one by one for each animation pair. Animazing already does it all automatically for you

Another example: A death animation can have a high priority, like 10, to beat all the others. An idle animation can have a low priority, such as 0, to be played only when nothing else is happening. You can set the priority values you want when playing, including fractional or negative values.

The Animazing Priority System can be used not only between animations, but also to decide disputes between competing behaviors (code being executed), as we will see later. This makes it easier to model a more realistic AI and an avatar with more details.

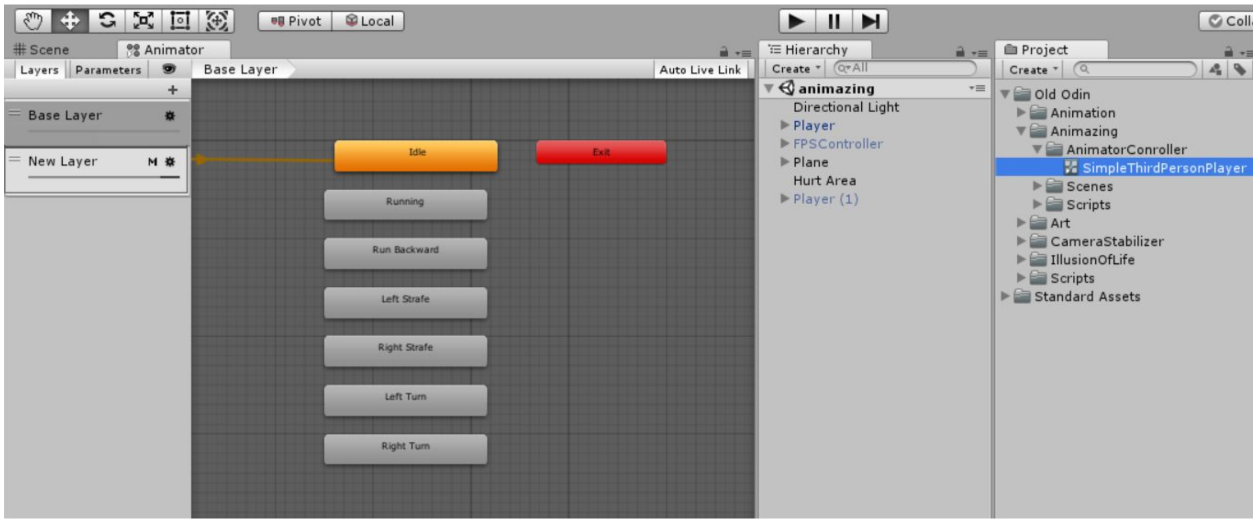
Animazing also offers other extra methods that make it easier to deal with rich animations and behaviors, which we list here.

3. A Quick Setup

With Animazing, the Animator Controller is quickly set up and used:

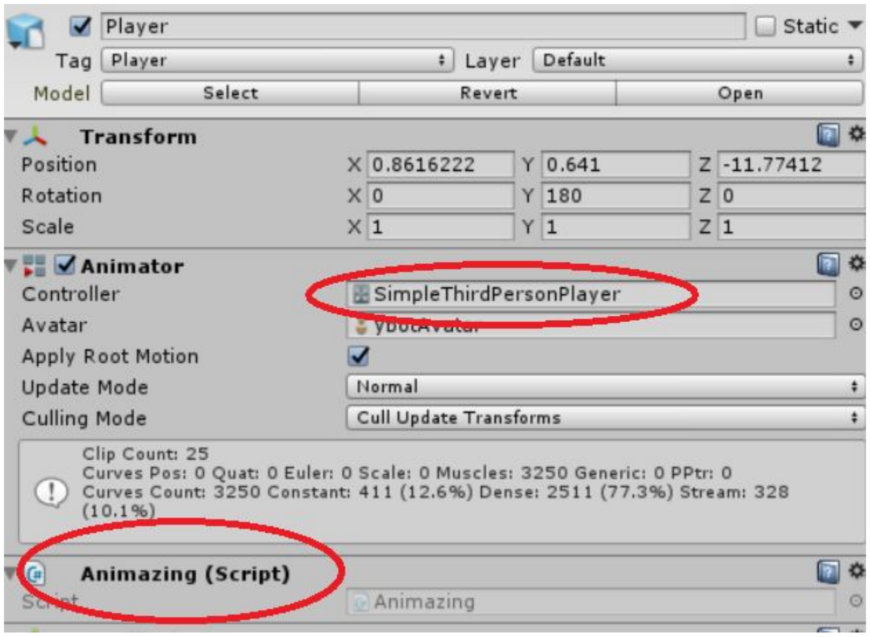
1) Create a new Animator Controller (By right-clicking in the Project View and selecting '**Create> Animator Controller**'.) And drag the animations you want to use onto it:

You do not need to create any kind of transition, sub-states, blend trees or any such complexity between states. Simply leave them listed there.



- ATTENTION: DO NOT rename the states inside the Animator Controller. Because they need to have the SAME name as the animation clip; If you want the state to have another name, rename the animation clip. Unity will automatically adjust the state name.

2) Place the Animator Controller in the Animator component of your character and also add the behavior Animazing to it.



Player ☐ Static
Tag: Player   Layer: Default
Model: Select | Revert | Open

Transform

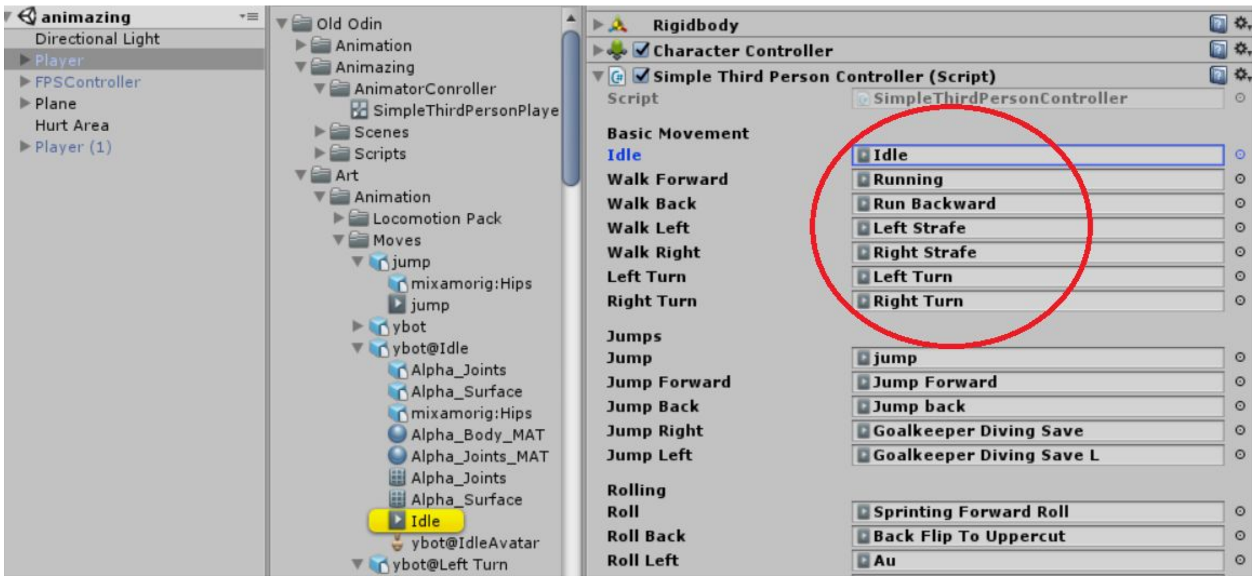
	X	Y	Z
Position	0.8616222	0.641	-11.77412
Rotation	0	180	0
Scale	1	1	1

Animator
Controller: SimpleThirdPersonPlayer
Avatar: ybotAvatar
Apply Root Motion: ✓
Update Mode: Normal
Culling Mode: Cull Update Transforms

```
Clip Count: 25
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 3250 Generic: 0 PPtr: 0
Curves Count: 3250 Constant: 411 (12.6%) Dense: 2511 (77.3%) Stream: 328
(10.1%)
```

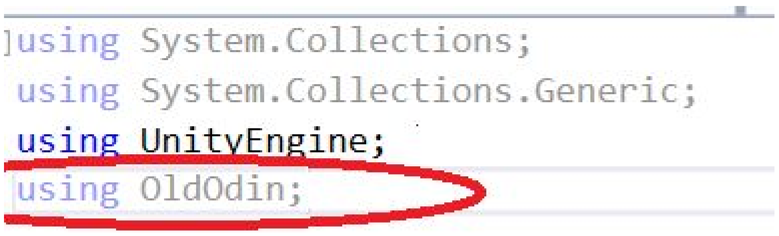
Animazing (Script)
Script: Animazing

3) In your character's script, create public variables of the type Animation Clip and put in them the animations you plan to use with appropriate names. .



Script	SimpleThirdPersonController
Basic Movement	
Idle	Idle
Walk Forward	Running
Walk Back	Run Backward
Walk Left	Left Strafe
Walk Right	Right Strafe
Left Turn	Left Turn
Right Turn	Right Turn
Jumps	
Jump	jump
Jump Forward	Jump Forward
Jump Back	Jump back
Jump Right	Goalkeeper Diving Save
Jump Left	Goalkeeper Diving Save L
Rolling	
Roll	Sprinting Forward Roll
Roll Back	Back Flip To Uppercut
Roll Left	Au

4) add “using OldOdin” to your class



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using OldOdin;
```

Done, the animazing code can now be used. Let's see below.

(PS: You don't need to be an Animator Controller expert to use Animazing, but you need to at least know the most basic concepts. Any questions, please consult: https://docs.unity3d.com/Manual/class-AnimatorController.html)

4. The Mandatory Default Animation

At Start and Awake, take the Animazing component and define the default animation for the base layer (usually an idle animation), like this:

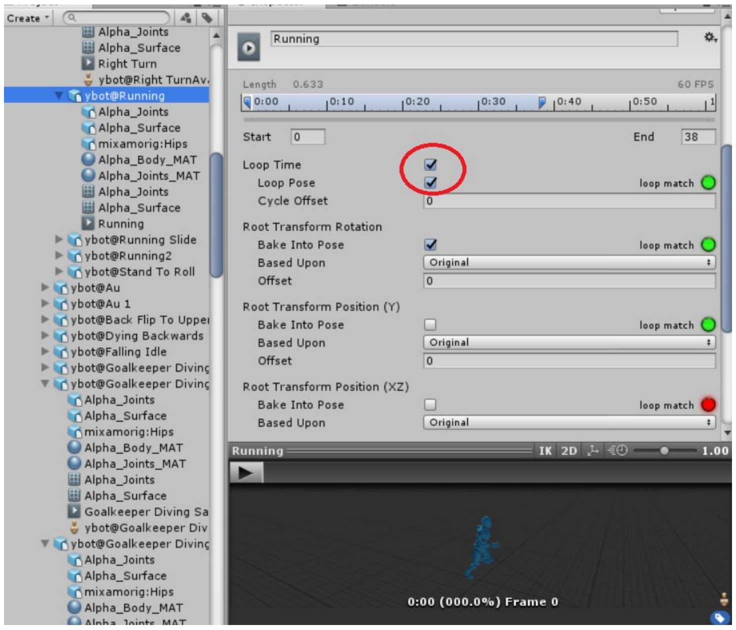
```
public AnimationClip idle;
Animazing animazing;

void Start()
{
    animazing = GetComponent<Animazing>();
    animazing.SetLayerDefaultAnimation(0, idle);
}
```

This animation will now play automatically whenever there are no other priority animations being called. It will also help with the transitions between the character's discrete actions. You are required to call this method at least once at the beginning of the script. If you later want to change the default idle animation, you can call this method again at any time. Default animation has priority - Infinity.

5. Loop Animations with Smart End

To test this functionality, do the following;

a) Mark an animation clip as "looping", for example, a floor animation.



Running

Length 0.633 60 FPS

Start 0 End 38

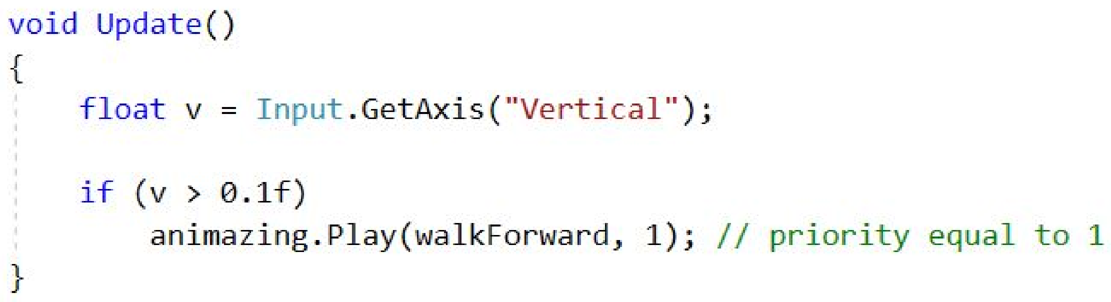
Loop Time ✓
Loop Pose ✓ loop match 🟢
Cycle Offset 0

Root Transform Rotation
Bake Into Pose ✓ loop match 🟢
Based Upon Original
Offset 0

Root Transform Position (Y)
Bake Into Pose loop match 🟢
Based Upon Original
Offset 0

Root Transform Position (XZ)
Bake Into Pose loop match 🔴
Based Upon Original

b) Call the Play method with a higher priority than - Infinity. For example:

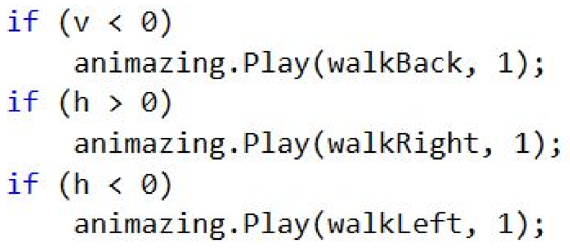


```
void Update()
{
    float v = Input.GetAxis("Vertical");

    if (v > 0.1f)
        animazing.Play(walkForward, 1); // priority equal to 1
}
```

Note that now when you press W your character smoothly exits the idle animation and starts moving forward. If you release the W, the avatar automatically returns to the idle animation, also with a smooth transition, without having to configure anything in the Animator Controller and, most importantly: WITHOUT you having to call any Stop method to stop the animation of walking. In comparison, using traditional Unity Play, the walk animation would loop indefinitely. Incidentally, the animator.Play can not even be called at every frame, or the animation will restart all the time. Why doesn't this happen with Animazing ?.

Animazing is a smart solution suited to the standards of game development. We understand that very often looped animations, such as pressing a key to walk, or the AI watching the player and chasing him, will need to be (re) validated for each frame. Therefore, every animation clip marked as a loop must be called for each frame in order to continue playing. If the clip is no longer called, the transition to the next active animation will start in order of priority (in this example, the Idle animation). However, if you need to, you can circumvent this feature if you add an extra duration time using the minDuration parameter, see the Play method parameters references.

You can easily add other motion animations where the player (or the AI) needs to be actively pressing a key or checking for conditions.

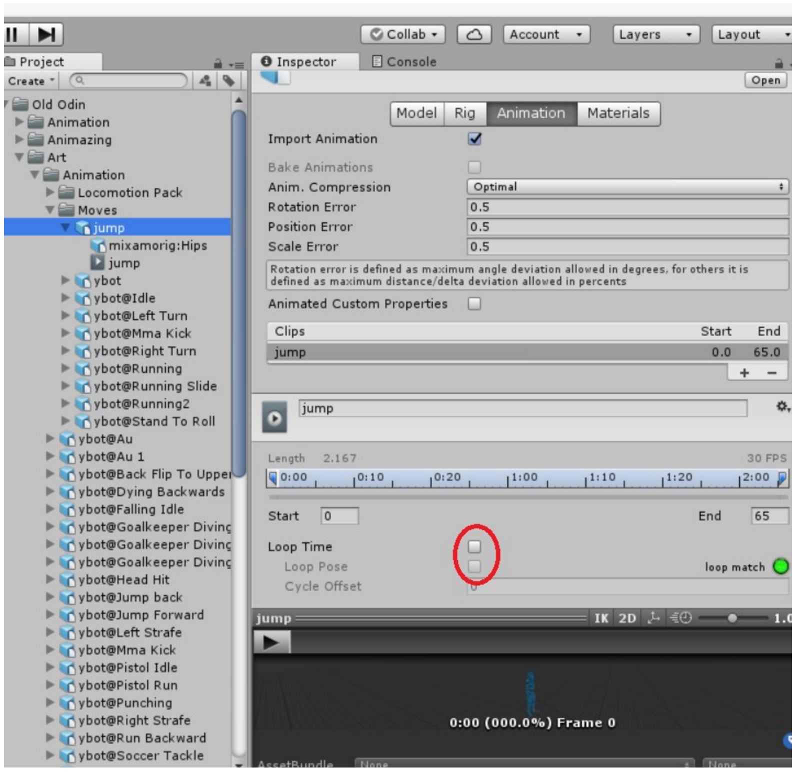


```
if (v < 0)
    animazing.Play(walkBack, 1);
if (h > 0)
    animazing.Play(walkRight, 1);
if (h < 0)
    animazing.Play(walkLeft, 1);
```

6. Action Animations and Tiebreaker

To play discrete actions (such as jumping, hitting, shooting, taking damage, etc.), do the following:

c) Make sure the animation clip is NOT marked as a loop.



Import Animation [✓]
Bake Animations [ ]
Anim. Compression: Optimal
Rotation Error: 0.5
Position Error: 0.5
Scale Error: 0.5
Rotation error is defined as maximum angle deviation allowed in degrees, for others it is defined as maximum distance/delta deviation allowed in percents
Animated Custom Properties [ ]

Clips	Start	End
jump	0.0	65.0

jump
Length 2.167 30 FPS
Start 0 End 65
Loop Time [ ]
Loop Pose [ ]
Cycle Offset 0
loop match

d) Play the animation at the appropriate time and with the appropriate priority.

Example:

```
if (Input.GetButtonDown("Fire1"))
    animazing.Play(punch, 2);
```

;

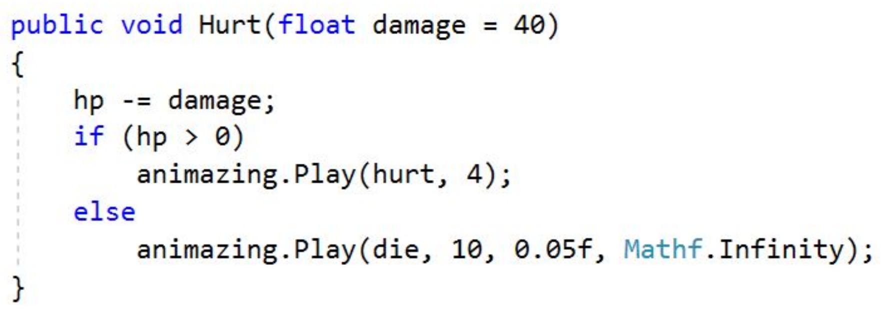
Here the avatar will display a punch animation that, as it has a priority greater than 1, will overwrite the floor animation, if the player is also pressing W.

While the punch animation is playing, any lower priority animation that tries to be played in that period will be automatically ignored. The animation will automatically stop at the end or if any animation with a priority higher than 2 interrupts it (such as taking damage, for example), without you having to do these checks.

When animations have the same priority, as in the example above, the first one to be called in time will have preference. Use the SetTiebreaker method if you want to reverse this tiebreaker preference and prioritize the last animations that were called.

7. Behavior Priority System

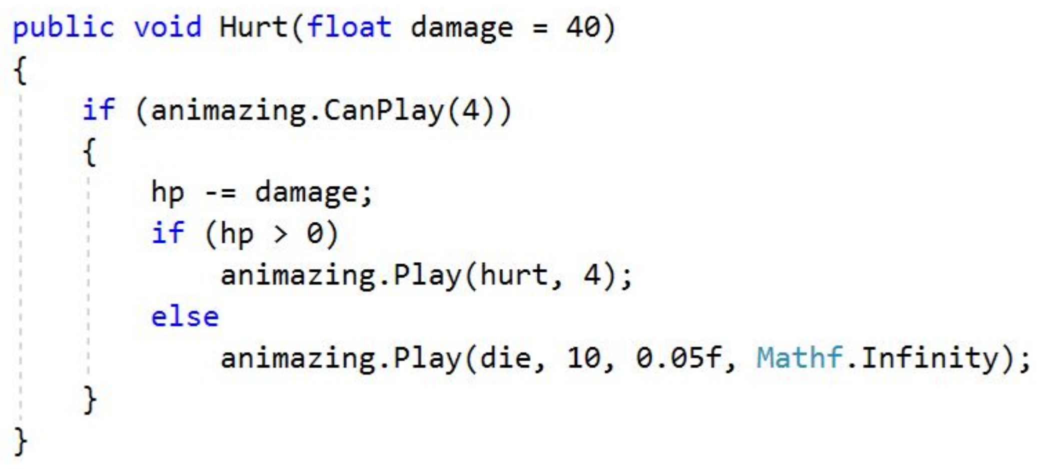
Often the same Priority System used to decide conflicts between animations can also be used to decide conflicts between behaviors (code being executed). For example,initially we could write a method to damage an agent like this:



```
public void Hurt(float damage = 40)
{
    hp -= damage;
    if (hp > 0)
        animazing.Play(hurt, 4);
    else
        animazing.Play(die, 10, 0.05f, Mathf.Infinity);
}
```

The agent has its total hp decreased, if the hp reached zero, we play the death animation, if not, we play the damage animation.

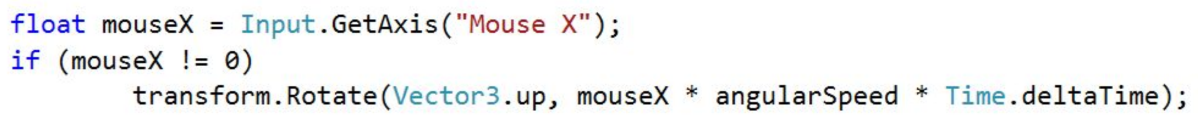
Notice that we are decreasing the hp variable even though we are not sure that the damage animation will play. It may be that some higher priority animation is playing and prevents the damage animation (For example, a defense animation, another previous damage animation still in progress, or even a death animation - we generally don't want the HP to keep decreasing, when the player is dead or while the player is suffering a previous damage). So, before decreasing the HP, we need to check if at that moment at least it is possible to play the damage animation. The CanPlay method is used for this:



```
public void Hurt(float damage = 40)
{
    if (animazing.CanPlay(4))
    {
        hp -= damage;
        if (hp > 0)
            animazing.Play(hurt, 4);
        else
            animazing.Play(die, 10, 0.05f, Mathf.Infinity);
    }
}
```

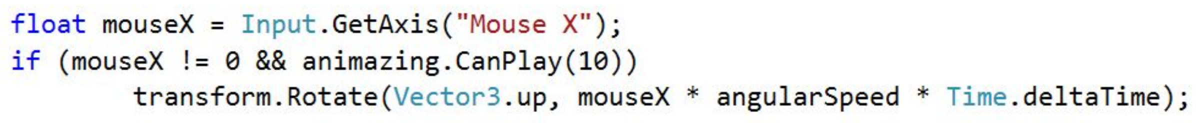
Being 4 the priority we chose for damage animation, what we are doing is checking if animations with this priority are likely to be played on the base layer at that moment, before executing the animation-related code.

This technique can be extended even to pieces of code where there is no animation being played, for example:



```
float mouseX = Input.GetAxis("Mouse X");
if (mouseX != 0)
        transform.Rotate(Vector3.up, mouseX * angularSpeed * Time.deltaTime);
```

Here the avatar is rotated with the horizontal movement of the mouse. But we don't want this behavior to continue after the player dies (we don't want the player to be able to roll the corpse). As our death animation has priority 10, we can write the following:

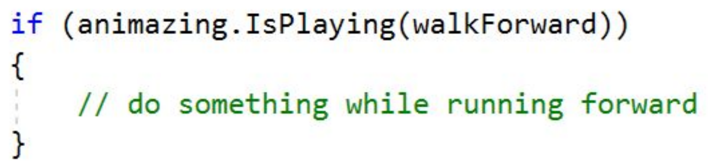


```
float mouseX = Input.GetAxis("Mouse X");
if (mouseX != 0 && animazing.CanPlay(10))
        transform.Rotate(Vector3.up, mouseX * angularSpeed * Time.deltaTime);
```

Thus, we can only roll if priority 10 (which has been reserved for states of death) is not being used.

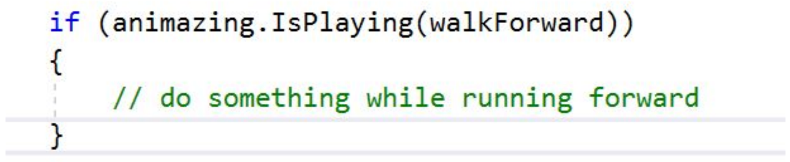
This technique can be used on many MonoBehaviours independently to model rich behavior and avoid conflict without making complex comparisons.

On the other hand, if you want to make sure that a given code will run ONLY while a given animation is playing, you can use the IsPlaying method



```
if (animazing.IsPlaying(walkForward))
{
    // do something while running forward
}
```

Or we can group all animations of walking at priority 1 and doing something when any of them are playing:



```
if (animazing.IsPlaying(walkForward))
{
    // do something while running forward
}
```

8. Stopping Animations

You may need to stop an animation in progress. For example, canceling an ongoing attack, or resurrecting an agent who was already dead, like a zombie. In this case you can use the Stop method:

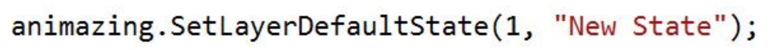
animazing.**Stop** (death);

The next priority animation will take over, probably the default idle animation.

9. Animations in other layers

Layer are useful if you want, for example, to create an aiming animation with a weapon in layer 1, which affects only the arms, while the base layer continues to walk and control the movement of the legs. Consult the Unity manual for more information. Animazing allows you to interact with the Animator Controller layers as follows:

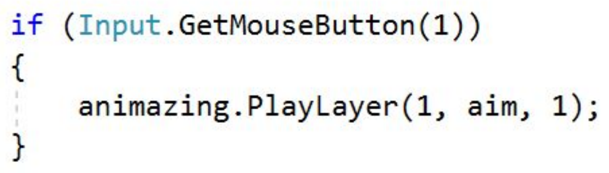
First, create an empty state for that layer (the default name that Unity gives to this state when it is created is “New State”, you can change the name if you want). Then you can do this to set this empty state as the default:



```
animazing.SetLayerDefaultState(1, "New State");
```

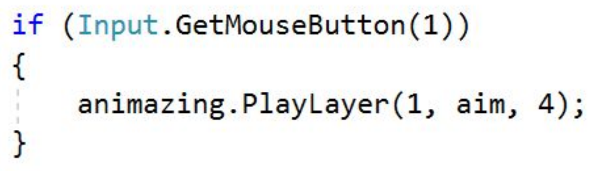
Use **SetLayerDefaultState** for empty states only. If you want to set an animation clip as the default, use **SetLayerDefaultAnimation**.

To play an animation on a top layer, use the PlayLayer method passing the layer index. The base layer is 0, the second layer is 1, the third is 2 and so on). For example:



```
if (Input.GetMouseButton(1))
{
    animazing.PlayLayer(1, aim, 1);
}
```

Each layer has its own Priority System that does not affect the previous layers, but is affected by them. For example, a priority 4 animation on the base layer can bar a priority 2 animation on layer 1, but a priority 4 animation on layer 1 cannot bar a priority 2 animation on the base layer. So, if we want the aim animation to be stopped only when the base layer executes a damage animation or another of even greater priority than the damage (4), we can write the code:



```
if (Input.GetMouseButton(1))
{
    animazing.PlayLayer(1, aim, 4);
}
```

In this way, the animations in progress on the lower layers greater than or equal to 4 will limit the aim animation when it trying to be played on the upper layer.

10. Script Reference

void **SetLayerDefaultAnimation**(int layer, AnimationClip defaultClip, float seconds = 0.2f)

- **layer:** The layer where the animation will be played. The animation clip must be on the corresponding layer in the Animator Controller. The base layer = 0
- **defaultClip**: default animation clip to be played Generally, an idle animation.
- **seconds**:Time seconds for new animation.

Defines the default animation that will be played on that layer when no other animation is playing. This animation will have the lowest priority of all: - Infinity.

---

void **SetLayerDefaultState**(int layer, string stateName, float seconds = 0.2f)

- **layer:** The layer where the state will be played. The state must be on the corresponding layer in the Animator Controller. The base layer = 0
- **stateName:** state to be played Generally, an empty state
- **seconds:** Time seconds for new state.

Defines the default state that will be played on that layer when no other animation is playing. This state will have the lowest priority of all: - Infinity.

---

void **Play**(AnimationClip clip, float priority, float seconds = 0.2f, float minDuration = 0)

- **clip**: Animation clip to be played.
- **priority**: Animation priority. If two animations are played at the same time, the animation with the highest priority will be displayed
- **seconds**:Time seconds for new animation It ranges from 0 to 1.
- **minDuration:** Minimum duration of the animation. Useful for avoiding interrupted gestures too early and for extending a looping animation.For death animations, use minDuration = Mathf.Infiniy.
- **layer:** The layer where the animation will be played. The animation clip must be on the corresponding layer in the Animator Controller. The base layer = 0

Plays the animation on the base layer while no higher priority animation is playing. Call this function at Start or Update. The winning animation will start playing on LateUpdate.

---

void **Play**Layer(int layer, AnimationClip clip, float priority, float seconds = 0.2f, float minDuration = 0)

- **clip:** Animation clip to be played
- priority: Animation priority. If two animations are played at the same time, the animation with the highest priority will be displayed.
- **seconds:** Time seconds for new animation;
- **minDuration:** Minimum duration of the animation. Useful for avoiding interrupted gestures too early and for extending a looping animation.For death animations, use minDuration = Mathf.Infiniy.
- **layer:** The layer where the animation will be played. The animation clip must be on the corresponding layer in the Animator Controller. The base layer = 0

Plays the animation on the specified layer while no higher priority animation is playing. Call this function at Start or Update. The winning animation will start playing on LateUpdate.

---

bool I**sPlaying**(AnimationClip clip, int layer = 0)

Check if the animation clip is currently being played on the layer. It only works for animations that were played using Animazing.

---

bool **IsPlaying**(float priority, int layer = 0)

Returns true if there is any animation being played on the layer with the given priority. Useful to know if a member of a set of animations with the same priority (such as attacks) is playing.

---

void **Stop**(int layer = 0)

Stops the animation currently being played on the layer. Useful to interrupt an action, for example. It only works for animations that were played using Animazing.

---

void **Stop**(AnimationClip clip, int layer = 0)

Stops the specified animation if it is currently being played on the layer. Useful to interrupt an action, for example.It only works for animations that were played using Animazing.

---

bool **CanPlay**(float priority, int layer = 0)

Returns true if the animation currently being played on the layer has a lower priority than the specified priority. Useful for performing certain behaviors only when the situation allows.

---

void SetTiebreaker(float priority, TiebreakerCriteria tiebreakerCriteria, int layer = 0)

- **priority**: The priority that will be affected.</param>
- **tiebreakerCriteria:** Tiebreaker criteria: in favor of the oldest animations, or in favor of the most recent animations.
- **layer:** The layer that will be affected.

Sets the tiebreaker for animations with the same priority. The default value favors older animations over newer ones.

---

float **GetPriority**(int layer = 0)

returns the current priority of the base layer or the specified layer