# Anim tutorial

## Summary

See previous basic tutorials for more info about basic object creation, clock handling and frames.

This tutorial only covers the very basic use of animations in orx.

All animations are stored in a directed graph.
This graph defines all the possible transitions between animations. An animation is referenced using a unique character string. All the transitions and animations are created via config files.

When an animation is requested, the engine will evaluate the chain that will bring it to this animation from the current one.
If such a chain exist, it will then be processed automatically. The user will be notified when animations are started, stopped, cut or looping by events.
If we don't specify any animation as target, the engine will follow the links naturally according to their properties [1].
There's also a way to bypass this chaining procedure and immediately force an animation.

Code-wise this system is very easy to use with two main functions to handle everything. Most of the work is made not in code but in the config files when we define animations and links. [2]

## Details

As usual, we begin by creating a viewport, getting the main clock and registering our `Update` function to it and, lastly, by creating our main object.
Please refer to the previous tutorials for more details.

Now let's begin by the code, we'll see how to organize the data at the end of this page.
In our `Update` function we'll just trigger a `WalkLeft` animation when the input `GoLeft` is active and a `WalkRight` animation when the input `GoRight` is active.
When no input is active, we'll simply remove the target animation and let the graph be evaluated naturally [3].

```
if(orxInput_IsActive("GoRight"))
{
  orxObject_SetTargetAnim(pstSoldier, "WalkRight");
}
else if(orxInput_IsActive("GoLeft"))
{
  orxObject_SetTargetAnim(pstSoldier, "WalkLeft");
}
else
{
  orxObject_SetTargetAnim(pstSoldier, orxNULL);
```

```
}
```

That's it! How to go from any current animation to the targeted one will be evaluated using the graph. If transitions are needed they'll be automatically played [4].

*NB: If we had wanted to go immediately to another animation without respecting data-defined transitions (in the case of hit or death animations, for example), we could have done this.*

```
orxObject_SetCurrentAnim(pstSoldier, "DieNow");
```

*NB: There are more functions for advanced control over the animations (like pausing, changing frequency, ...), but 99% of the time, those two functions (`orxObject_SetCurrentAnim()` and `orxObject_SetTargetAnim()`) are the only ones you will need.*

Let's now see how we can be informed of what happens with our animations (so as to synchronize sounds, for example).
First, we need to register our callback `EventHandler` to the animation events.

```
orxEvent_AddHandler(orxEVENT_TYPE_ANIM, EventHandler);
```

Done! Let's see what we can do with this now.
Let's say we want to print which animations are played, stopped, cut or looping on our object. We would then need to write the following callback.

```
orxSTATUS orxFASTCALL EventHandler(const orxEVENT *_pstEvent)
{
orxANIM_EVENT_PAYLOAD *pstPayload;

pstPayload = (orxANIM_EVENT_PAYLOAD *)_pstEvent->pstPayload;

switch(_pstEvent->eID)
{
  case orxANIM_EVENT_START:
    orxLOG("Animation <%s>@<%s> has started!", pstPayload->zAnimName,
orxObject_GetName(orxOBJECT(_pstEvent->hRecipient)));
    break;

  case orxANIM_EVENT_STOP:
    orxLOG("Animation <%s>@<%s> has stopped!", pstPayload->zAnimName,
orxObject_GetName(orxOBJECT(_pstEvent->hRecipient)));
    break;

  case orxANIM_EVENT_CUT:
    orxLOG("Animation <%s>@<%s> has been cut!", pstPayload->zAnimName,
orxObject_GetName(orxOBJECT(_pstEvent->hRecipient)));
    break;

  case orxANIM_EVENT_LOOP:
    orxLOG("Animation <%s>@<%s> has looped!", pstPayload->zAnimName,
orxObject_GetName(orxOBJECT(_pstEvent->hRecipient)));
```

```
    break;
  }

  return orxSTATUS_SUCCESS;
}
```

We first get the payload of our event. As we know we only handling animation events here, we can safely cast the payload to the `orxANIM_EVENT_PAYLOAD` type defined in orxAnim.h.

If we were using the same callback for different event types, we first would need to see if we were receiving an anim event. This can be done with the following code.

```
if(_pstEvent->eType == orxEVENT_TYPE_ANIM)
```

Finally, our event recipient (`_pstEvent→hRecipient`) is actually the object on which the animation is played. We cast it as a `orxOBJECT` using the helper macro `orxOBJECT()`. [5)]

Let's now have a peek a the data side.

First, so as to reduce the amount of text we need to write, we'll use orx's config system inheritance. We'll begin to define a section for the position of our pivot [6)].
As you may have seen in the object tutorial config file, the pivot is which position will match the world coordinate of your object in the world space. If it's not specified, the top left corner will be used by default.
The pivot can be defined literally using keywords such as `top`, `bottom`, `center`, `left` and `right`, or by giving an actual position, in pixels.

```
[Pivot]
; This define the pivot we will use for all our animation frames
Pivot = (15.0, 31.0, 0.0)
```

Next, we'll define our graphic object that will inherit from this pivot. In our case it's a bitmap that contains all the frames for our object.
The common properties are thus the name of the bitmap file and the size of one frame [7)].

```
[Graphic@Pivot]
; This is the main graphic object, used when no animation is played
Texture = soldier.png
```



The optional `Graphic` property is only used to supply a default frame size for the animation frames. If you don't define this, the first frame from your animation will supply the frame size.

Last, we need to define an animation set that will contain the whole graph for our specific object's animations.
The animation set won't ever be duplicated in memory and will contain all the animations and links for the corresponding graph.

In our case we have 4 animations and 10 possible links for all the transitions.

```
[Soldier]
Graphic             = Graphic
AnimationSet        = AnimSet
Scale               = 4.0


[AnimSet]
Direction   = right # down
StartAnim   = IdleRight
KeyDuration = 0.1
Digits      = 1
FrameSize   = (32, 32, 0)
Texture     = soldier_full.png ; <= This property will be inherited by all
the frames
Pivot       = @Pivot ; <= This property will be inherited by all the frames
IdleRight   = 1 ; <= We only want one frame
IdleLeft    = 1
WalkRight   = -1 ; <= We want as many frame that can fit in the texture
defined by WalkRight
WalkLeft    = -1
IdleRight-> = IdleRight # .IdleLeft # .WalkRight ; <= When going from
IdleRight to IdleLeft, the IdleRight animation will be interrupted
IdleLeft->  = IdleLeft # .IdleRight # .WalkLeft
WalkRight-> = WalkRight # .+IdleRight ; <= When going from WalkRight to
IdleRight, the WalkRight animation will be interrupted and if no animation
target is defined, WalkRight will always lead to IdleRight
WalkLeft->  = WalkLeft # .+IdleLeft
```

Now to define some properties for the left-oriented animations. Actually as we're using flipped graphic objects, we could just have flipped the object at runtime in the code.
But that wouldn't have served our didactic purposes! Let's pretend these left animations are completely different from the right ones! 

```
[IdleLeft]
Flip        = x

[WalkLeft]
Flip        = x

[IdleRight]
Direction   = left # up

[IdleLeft]
Direction   = left # up

[WalkRight1]
KeyEvent    = !!Left!! ; Adds left foot event on 1st frame of WalkRight
```

```
[WalkRight4]
KeyEvent    = !!Right!! ; Adds right foot event on 4th frame of WalkRight

[WalkLeft1]
KeyEvent    = !!Right!! ; Adds right foot event on 1st frame of WalkRight

[WalkLeft4]
KeyEvent    = !!Left!! ; Adds left foot event on 4th frame of WalkLeft
```

*NB: This is a very basic graph that shows only basic transitions, but the system is very expandable. Let's say you want to begin walking from a sitting pause without transition.*
*But, later in the game development, you want to add a standing up transition for it to look nicer. You'll only have to add this extra step (with the associated links) in the config file AnimSet! Your code will remain unchanged:*

```
orxObject_SetTargetAnim(MyObject, "Walk");
```

## Resources

Source code: 04_Anim.c

Config file: 04_Anim.ini

1)
such as loop counters that won't be covered by this basic tutorial
2)
A very basic animation graph will be used for this tutorial: we did it so as to keep limited the amount of needed config data.
3)
in our case, it'll play the corresponding idle animation even if our data only contains one single frame for each
4)
remember that in our case we went for the straightest path, with no turning animations, for example, but that wouldn't change our code at all!
5)
Remember that we use such macros so as to make sure we're casting in the right type.
6)
also called `HotSpot` in some engines
7)
we don't have to keep it constant, but usually it's easier for artists and it's even a constraint for some other engines/libraries

From:
https://www.orx-project.org/wiki/ - **Orx Learning**

Permanent link:
**https://www.orx-project.org/wiki/en/tutorials/animation/anim?rev=1598876885**

Last update: **2025/09/30 17:26 (14 months ago)**