Tutorial de Animaciones

Resumen

Veasé los anteriores tutoriales básicos para más información acerca de creación básica de objetos, manejo del reloj y fotogramas.

Este tutorial solo cubre un uso muy básico de las animaciones en orx.

Todas las animaciones son guardadas en grafo dirigido.
Este gráfico define todas las posibles transisiones entre animaciones. Una animación es referenciada usando un único caracter de cadena. Todas las transiciones y animaciones son creadas via ficheros de configuración.

Cuando se solicita una animación, el motor evaluará la cadena que lo traerá a esta animación desde la actual.
Si esa cadena existe, a continuación se procesará automáticamente. El usuario será notificado cuando se inician las animaciones, se detienen, se corten o estén en un bucle de eventos.
Si no especificamos ninguna animación como objetivo, el motor va a seguir los enlaces naturalmente, de acuerdo a sus propiedades [1)].
También hay una manera de saltarse este procedimiento de encadenamiento y forzar una animación inmediatamente.

El Código-sabio de este sistema es muy fácil de usar con dos funciones principales para manejar todo. La mayoría del trabajo se realiza no en el código, sino en los archivos de configuración cuando se definen las animaciones y enlaces. [2)]

Detalles

Como de costumbre, comenzamos por la carga de nuestro archivo de configuración, creando una ventana, obteniendo el reloj principal y registrando nuestra función `Update` a la misma y, por último, mediante la creación de nuestro objeto principal.
Por favor, consulte los tutoriales anteriores para más detalles.

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 stoped!", 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 even 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, 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.

```
[AnimSet]
AnimationList = IdleRight#WalkRight#IdleLeft#WalkLeft

LinkList =
IdleRightLoop#IdleRight2Left#IdleRight2WalkRight#WalkRightLoop#WalkRight2Idl
eRight#IdleLeftLoop#IdleLeft2Right#IdleLeft2WalkLeft#WalkLeftLoop#WalkLeft2I
dleLeft
```

Now let's define our animations! 

Previous to that, 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]
Pivot = (15.0, 31.0, 0.0)
```

Now 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)].

```
[FullGraphic@Pivot]
Texture     = ../../data/anim/soldier_full.png
TextureSize = (32, 32, 0)
```

Ok, everything is setup for creating all the frames.
Let's define all our frames for both right-oriented animations: we have 6 of them.

```
[AnimRight1@FullGraphic]
TextureCorner = (0, 0, 0)

[AnimRight2@FullGraphic]
TextureCorner = (0, 32, 0)

[AnimRight3@FullGraphic]
TextureCorner = (0, 64, 0)

[AnimRight4@FullGraphic]
TextureCorner = (32, 0, 0)

[AnimRight5@FullGraphic]
TextureCorner = (32, 32, 0)

[AnimRight6@FullGraphic]
TextureCorner = (32, 64, 0)
```

As you can see, they all inherit from `FullGraphic` and the only property that tells them apart is the `TextureCorner`.

Ok, now we have defined all those graphic objects (they'll become `orxGRAPHIC` structures when loaded), let's define the animations themselve.
Let's begin with the `IdleRight` which contains a single frame that lasts for 0.1 second.

```
[IdleRight]
KeyData1      = AnimRight6
KeyDuration1  = 0.1
```

Easy enough, let's try the second one: `WalkRight`

```
[WalkRight]
DefaultKeyDuration  = 0.1
KeyData1            = AnimRight1
KeyData2            = AnimRight2
KeyData3            = AnimRight3
KeyData4            = AnimRight4
KeyData5            = AnimRight5
KeyData6            = AnimRight6
```

Not really harder as we define the same time for all the frames using the `DefaultKeyDuration` property.
We can override it for any frame by specifying a key duration like we did for the `IdleRight` animation.

We'll do the exact same thing 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! 

There are only the links missing now and we're done!
The basic link structure is easy: we specify the source and the destination animation.

```
[IdleRightLoop]
Source      = IdleRight
Destination = IdleRight
```

This link goes from `IdleLoop` animation to the `IdleLoop` animation. No wonder we called this link `IdleRightLoop`!
So basically, when we are in the `IdleRight` animation and we asked `IdleRight` animation as a target, we just stay there, looping.
Also, if no target is defined when we are there, this link will keep us looping as there isn't any higher priority link starting from `IdleRight`.

Let's see now how we go from `IdleRight` to `WalkRight`.

```
[IdleRight2WalkRight]
Source      = IdleRight
Destination = WalkRight
Property    = immediate
```

Here we have the same basic info as before but we also have the `immediate` value for the key `Property`.
This means that when we are in `IdleRight` animation and we target `WalkRight`, we won't wait till `IdleRight` is over to follow the link, we'll go there directly: that gives us a way to cut animations.

As we've seen in the code, we don't explicitely ask for the idle animation when we are already walking. How do this work then??
Let's see the link that goes from `WalkRight` to `IdleRight`.

```
[WalkRight2IdleRight]
Source      = WalkRight
Destination = IdleRight
Property    = immediate; <= If you remove this property, the animation won't
be cut to go immediately back to idle
Priority    = 9
```

When we are in `WalkRight` and we remove our target, the engine will have to follow the links in a natural way. That means it'll favorise the higher priority links.
By default a link `Priority` is 8. It can range from 0 to 15. Having here a link at priority 9 means that

this will be the one taken when we have no target.
It will bring us back to `IdleRight`.
We also have added the `immediate Property` so that we won't wait the end of the walk cycle to go back to idle.

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! Your code will remain unchanged:

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

Recursos

1)
tales como contadores de bucles que no serán cubiertos por este tutorial básico
2)
Un gráfico de animación muy básica será utilizada para este tutorial: lo hicimos a fin de mantener limitada la cantidad de datos de configuración necesarios.
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 event a constraint for some other engines/libraries

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

Permanent link:
https://www.orx-project.org/wiki/es/orx/tutorials/anim?rev=1330532647

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